NASA Contractor
Report No. 145298

Report No.
ATR-78(7640)-1

FAULT-TOLERANT SOFTWARE STUDY

Advanced Programs Division
THE AEROSPACE CORPORATION
El Segundo, California 90245

1 February 1978

Final Report

Prepared for

Langley Research Center
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Hampton, Virginia 23365

Contract NAS1-14644

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
ATR-78(7640)-1		

4. Title and Subtitle	5. Report Date
Fault-Tolerant Software for Aircraft Control Systems	1 February 1978
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
	10. Work Unit No.
9. Performing Organization Name and Address The Aerospace Corporation El Segundo, California	11. Contract or Grant No. NAS1-14644
12. Sponsoring Agency Name and Address NASA Langley Research Center Hampton, Virginia	13. Type of Report and Period Covered Final Report Oct 1966 - Oct 1977
	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract This report addresses concepts for software to implement real-time aircraft control systems on a centralized digital computer. A "fault-tolerant" software structure employing functionally redundant routines with concurrent error detection, and provisions to switch from one routine to a functional alternate in the event of a detected fault, after the recovery block structure conceived by Randell, et al, at the Computing Laboratory, University of Newcastle-upon-Tyne, United Kingdom, is proposed for critical control functions involving safety of flight and landing. A "degraded" recovery block concept, incorporating concurrent error detection with an abort return but no functional alternate routine, is devised to allow collocation of critical and non-critical software modules within the same control structure. The degraded recovery block reduces the likelihood that failures in the non-critical modules will hang up the computer and thus deny access to the critical functions. Control resides in the task scheduler of the executive software, and, being a critical software module, it too is configured after the recovery block with a primary and functionally alternate routine. To guard against correlated failures, functionally alternate routines within a recovery block should be of different design. Conceptually, this requirement is easily achieved in aircraft flight controls through the use of either direct measurements, or variables derived from other functions. The additional computer resources required to implement the proposed software structure for a representative set of aircraft control functions is discussed. It is estimated that approximately 30% more memory space is required to implement the total set of control functions, compared to that required for non-fault-tolerant software, if only the flight-critical functions are implemented in the full recovery block format. A reliability model for the fault-tolerant software is described and parametric estimates of failure rate are made for various assumptions regarding failure probabilities for the component software modules.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Fault-tolerant software Reliable software Digital aircraft control Fault-tolerant computing Recovery block structured software	

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*

* For sale by the National Technical Information Service, Springfield Virginia 22161

NASA Contractor
Report No. 145298

Aerospace Report No.
ATR-78(7640)-1

FAULT TOLERANT SOFTWARE STUDY (U)

Approved

G. Wm. Anderson, Group Director
Development Group Directorate
Advanced Programs Division

ACKNOWLEDGMENTS

The work reported here was performed by The Aerospace Corporation under Contract NAS1-14644 with the NASA Langley Research Center under the technical guidance of Mr. G. E. Migneault. Many helpful suggestions for the conduct of this study were received from him and from his colleagues S. Bavuso and N. Murray.

The principal investigator on this study was Dr. Herbert Hecht. Dr. V. B. Schneider designed the primary and alternate scheduler and directed the simulation, which was implemented by R. Noke and A. Sylvain.

Much valuable assistance was received from Ms. Bonnie Callender, Ms. Bonnie J. Schmidt and Ms. Phyllis Whobrey, in the preparation of the illustrations and manuscript, and in the typing of the final copy.

CONTENTS

ACKNOWLEDGMENTS i

1. INTRODUCTION 1

2. CONCEPTS FOR FAULT TOLERANT SOFTWARE 7

3. FAULT TOLERANT SOFTWARE STRUCTURE FOR REAL-TIME AIRCRAFT CONTROL 12

4. RESOURCE REQUIREMENTS FOR FAULT-TOLERANT SOFTWARE 29

5. FAULT-TOLERANT SCHEDULER AND FLIGHT-CONDITION MODULE ... 38

6. RELIABILITY OF FAULT TOLERANT SOFTWARE 54

7. CONCLUSIONS 63

REFERENCES 65

APPENDIX A - FLOW OF PASCAL SCHEDULER 68

APPENDIX B - SCHEDULER SIMULATION 69

GLOSSARY OF SOFTWARE HIERARCHY TERMS 74

FIGURES

1. A Simple Recovery Block 8

2. A More Complex Recovery Block 8

3. Parallel Processes with Conversations 11

4. Fault Tolerance for Application Modules 15

5. Internal Structure for Primary Application Module.... 21

6. Structure for Fault-Tolerant Aircraft Control Software..... 26

7. Critical Function Backup 28

8. Primary Scheduler 42

9. Alternate Scheduler 44

10. Fault-Tolerant Scheduler 47

11. Major-Minor Cycle Scheduling 49

12. Transition Model for an Application Routine..... 55

B-1 Scheduler Simulation Logic 70

B-2 Simulation Trace of FTS 72

B-3 Principal Sources of Messages in the Simulation Trace..... 73

v

TABLES

1. Analysis of Aircraft.Control Computational Requirements..... 31

2. Explanation of Criticality Levels..... 33

3. Impact of Fault Tolerant Provisions on Computation Requirements..... 35

4. Comparison of Primary and Backup Scheduler. Implementation..... 43

5. Sensor Interaction for Flight Condition Module..... 53

6. Failure Probability of a Fault-Tolerant Application Program..... 60

7. Failure Probability of Fault-Tolerant Software (Five Critical Programs)..... 61

Section 1

INTRODUCTION

This report summarizes the work performed at The Aerospace Corporation on a study of fault-tolerant software for the Langley Research Center, National Aeronautics and Space Administration, under Contract NAS1-14644. The objective of the study was to carry specific techniques for developing software systems which are tolerant of faults - within the software system as well as tolerant of faults within the hardware system into which the software system is embedded - closer to realization in real-time aircraft control systems. Specific techniques studied were those proposed by Professor B. Randell, et al, of the Computing Laboratory of the University of Newcastle-upon-Tyne, United Kingdom (Ref. 1). Extensions of Randell's concepts, to incorporate execution time as an essential acceptance test, to provide for an abort return, and to provide a "degraded" recovery block structure for non-critical functions, were included in the application studied under this contract. The functional and computation requirements of the real-time aircraft control application, which formed the frame of reference for this study, were those estimates prepared by the Stanford Research Institute under contract to NASA/Langley as part of a design study of a fault-tolerant airborne digital computer to be used as a central computer in an advanced, high performance commercial aircraft. (Ref. 2).

Automatic flight control concepts have evolved from simple pilot-relief autopilots to include sophisticated stability augmentation, gust and maneuver load alleviation, flutter control, energy management, and attitude and flight-path control from takeoff to touchdown. Although military applications have inspired much of the development in the past, commercial requirements for efficiency and economy, particularly for fuel economy, are rapidly becoming a major influence. In a recent paper (Ref. 3) it was noted that major improvements in aircraft performance and reductions in aircraft weight appear possible through combinations of currently independent aircraft functions such as active airframe control, propulsion control, landing loads control, and fuel management. The authors stated, as examples, that the integration of active landing gear and maneuver load control systems can appreciably decrease wing structural stiffness requirements and weight, and that automatic reconfiguration of control system gains in the event of an engine failure can allow sizeable reductions in required control surface areas. The authors assert that extension of this approach to fully-integrated, control-configured aircraft could provide up to 15 percent fuel savings and structural weight reductions. But, in conclusion, they observe that the integrity of the flight control system will continue to be the key factor in the acceptance of these concepts for operational application.

Federal Aviation Regulations (Ref. 4) state that airplane systems and associated components, considered

separately and in relation to other systems, must be designed so that:

(1) The occurrence of any failure condition which would prevent the continued safe flight and landing of the aircraft, or which, in the event of loss of all propulsive power, would preclude controlled flight to an emergency landing, is extremely improbable, and

(2) The occurrence of any other failure condition which would significantly reduce the operational or performance capability of the airplane is improbable.

In a draft FAA Advisory Circular (Ref. 5) it is explained that

(a) extremely improbable refers to occurrences expected with a mean frequency on the order of $1x10^{-9}$ or less per flight or flight hour, or occurrences so unlikely to occur that they need not be considered.

(b) improbable refers to occurrences which may be expected with a mean frequency in the approximate range of $1x10^{-5}$ to $1x10^{-9}$ per flight or flight hour, or occurrences not expected during the operation of an individual airplane, but expected to occur during the operational life of all airplanes of a type.

These are by no means trivial requirements. As a data point, the Air Force Space and Missile Systems Organization is supporting development, for use in unmanned spacecraft deployed in the late 1980's, of a fault tolerant

general purpose digital computer with an equivalent failure probability on the order of 10^{-6} per hour. At the system level to this failure probability must be added the equivalent failure probability of the input/output devices (sensors and actuators) and, of course, the software. Clearly, the achievement of failure probabilities of approximately 10^{-9} per flight hour for flight-critical systems for the next generation of aircraft is a major challenge; and perhaps just as challenging is the problem of demonstrating that such a failure rate has indeed been achieved once that point has been reached.

The software fault tolerance techniques considered in this study are similar in principle to those being applied to achieve fault tolerance in computer hardware, i.e., standby or "protective" redundancy, in the form of alternate hardware modules accessible by switching, and concurrent error detection. Provisions to permit rollback of the executing software to an uncontaminated location to recover from a failure are required in both instances. However, for hardware fault tolerance, the backup modules are usually identical in design to the primary module, but an identical copy of a computer program can hardly be expected to be of much help in recovering from a failure in the original. Therefore, redundancy in fault-tolerant software requires programs that are deliberately different from the original ones which they are to backup. In the aircraft control context, alternate

sensors, and perhaps alternate "actuators", primarily available to implement other functions, often can be employed to effect end-to-end independence of the alternate module design and hence lessen the probability that the backup fails under the same conditions as the primary module.

The redundancy and error detection provisions necessary for fault tolerance will of necessity involve additional hardware costs (e.g., by requiring additional memory) and performance penalties because more code has to be processed for a given task. However, if the fault tolerance provisions are incorporated only into those software modules that implement flight-critical functions, as defined in the SRI report (Ref. 2), the additional resources required to have essential fault tolerant provisions can be minimized.

With this as background, a software structure incorporating redundant fault-tolerant provisions for flight critical applications modules, non-redundant modules with error detection and flagging for non-critical functions, and a redundant fault-tolerant task scheduler, has been defined. The additional resources required to implement this structure as contrasted to the computational requirements set forth by SRI (Ref. 2) have been estimated. Employing a simplified reliability model parametric estimates of the failure probability of the resulting structure have been made using assumed values for failure rates for the component modules.

Use is made of multiple-sensed control parameters for the design of backup modules as opposed to multiple copies of the same sensor type. The executive task scheduler includes an alternate backup, and a flight condition module employing multiple-sensed data is incorporated to effect acceptance tests.

This report consists of one volume. A brief review of the fault-tolerant software concepts proposed by Randell, et al, follows this Introduction. Subsequent sections of the report describe the adaptation of these concepts to the aircraft control tasks, the resource requirements to implement this proposed fault-tolerant software for real-time aircraft control systems, the design of the fault tolerant scheduler and flight condition module, and finally, the expected reliability of the proposed structure under various assumed component failure probabilities. The Appendices include the code, written in PASCAL, for the scheduler and a description of the simulation used to test the scheduler code.

Section 2

CONCEPTS FOR FAULT TOLERANT SOFTWARE

The basic concepts for fault-tolerant software investigated in this study are those set forth by Prof. B. Randell, et al, of the Computing Laboratory, University of Newcastle upon-Tyne, U.K. (Ref. 1). The following paragraphs contain a synoptic review of those concepts to provide a frame of reference for subsequent discussion of their adaptation to real-time software for aircraft control systems. For a more complete presentation of these concepts, refer to Reference 1 and the further references contained therein. Figures 1, 2 and 3 are taken from Reference 1.

Briefly, Randell, et al, conceive a computer program to be structured of blocks, with the blocks consisting of alternative sequences of operations, i.e., primary and one or more alternates. Extra information is provided to the block to permit a determination of completion and acceptability (i.e., an acceptance test) of the result of a sequence, with rollback and transfer to an alternate sequence in the event of failure to pass the acceptance test (e.g., by exceeding a time limit to complete or by exceeding the expected range for non-local variables). Randell, et al, refer to this single-entry, single-exit software element as the "recovery block". Figure 1 is a schematic representation of the simple recovery block.

```
A:  ensure AT
    by      AP : begin
                     <program text>
               end
    else by AQ : begin
                     <program text>
               end
    else error
```

FIG. 1 A SIMPLE RECOVERY BLOCK

```
A:ensure AT
  by      AP:begin declare Y
                  <program text>
          B:ensure BT
            by      BP:begin declare U
                           <program text>
                    end
            else by BQ:begin declare V
                           <program text>
                    end
            else by BR:begin declare W
                           <program text>
                    end
            else error
            <program text>
          end
  else by AQ:begin declare Z
                  <program text>
          C:ensure CT
            by      CP:begin
                           <program text>
                    end
            else by CQ:begin
                           <program text>
                    end
            else error
          D:ensure DT
            by      DP:begin
                           <program text>
                    end
            else error
          end
  else error
```

FIG. 2 A MORE COMPLEX RECOVERY BLOCK

Alternate sequences within a recovery block may contain, nested within themselves, further recovery blocks as shown in Figure 2. Noting that all software faults result from design errors, Randell, et al, require that the alternate sequences or components be not merely copies but independent designs, so that there exists a reasonable likelihood that at least one of the alternates can cope with the circumstances causing the primary component to fail.

The acceptance test is to ensure that the operations performed by the recovery block satisfy the program that called the block. Hence, the acceptance test is performed by reference to the variables accessible to the recovery block rather than variables local to the block. Local variables can have no effect or significance after exit from the block; but, more important, the alternate components within the block will probably have different sets of local variables. The program calling the block may be capable of proceeding with any of a number of possible results from the operations and the acceptance test must establish that the results lie within this range of acceptability without regard for which alternate generates them.

Before entering an alternate component the process must be rolled back or restored to the state that existed before entry of the primary component (or preceding alternate,

if one exists). Only non-local variables, and in particular non-local variables that have been modified by the preceding process, have to be reset to rollback to the entry state. To effect this automatically and thus relieve the programmer of the error-prone task of explicit preservation of restart information, Randell, et al, employ what is termed a "recursive cache" to save non-local variables, just before they are modified. This is accomplished in real time by detecting assignments to non-local variables, and in particular by recognizing when an assignment to a non-local variable is the first to have been made to that variable within the current alternate. Related cache entries are discarded as recovery blocks are successfully completed.

To provide for recoverability under circumstances involving processes proceeding in parallel, which at the same time become mutually dependent by virtue of their interactions, Randell, et al, invoke a structure which is termed a "conversation" (see Figure 3). A recovery block that spans two or more processes is termed a conversation. The conversation serves to restrict progress of interacting processes in the interest of preserving recoverability by requiring all processes in conversation to satisfy their respective acceptance tests before any one of the processes may proceed. It is possible for processes to enter a conversation at differing times; but all of the processes must leave the

conversation together to ensure that none have purged their recovery or rollback data until all have passed their acceptance tests. Finally, for multi-level systems with virtual machine interfaces, Randell, et al, require that the interfaces be arranged so the higher level need not furnish support for control or error handling of a lower level - i.e. levels are separated by "opaque virtual machine interfaces".

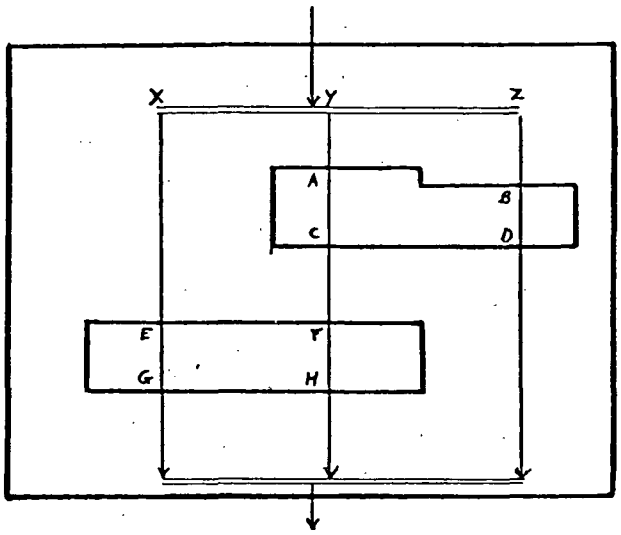


FIG. 3 PARALLEL PROCESSES WITH CONVERSATIONS (which provide recovery blocks for local communication)

Section 3

FAULT-TOLERANT SOFTWARE STRUCTURE FOR REAL-TIME AIRCRAFT CONTROL SYSTEMS

Given the basic structures described in the preceding section, the following discussion indicates how they might be organized to implement real-time aircraft control systems. It has been assumed that the system application involves the implementation of various closed-loop and open-loop functions (e.g., stability augmentation, flutter control, area navigation, energy/cruise management, automatic landing, etc.) in a centralized digital computer. The frequency with which these functions will have to be serviced will vary widely from one function to another; and for any one function, will vary depending upon the conditions of flight. However, it is presumed that these servicing requirements can be predicted and hence scheduled in time and for the anticipated flight conditions. Finally, here and in the rest of this report, it is assumed that the host computer is a multi-processor or multi-programming uniprocessor with a fault-tolerant architecture and recovery provisions that mask hardware failures.

Basic Software Structure

The simple control requirements (i.e., test and alternate routing) of the recovery block and the applicability

of the same general control structure to all recovery blocks is an important attribute of the fault tolerant concept that fits nicely into a multi-programming software structure. It permits the control features to be incorporated in the executive software, specifically in the task scheduler, rather than being replicated in every application block. Hence, the control structure can conceivably be tested so exhaustively that the likelihood of failure of this element of the software can be ruled out. With control of the recovery block resident in the scheduler, task synchronization and prioritization is more readily effected.

Since this is a time-shared operation, and timely servicing of many aircraft control functions is a necessity to meet accuracy and stability criteria, the time required to execute each application sequence is considered to be an essential acceptance test criteria for all blocks. In the fault-tolerant software structure proposed here this is accomplished by a watchdog timer, a special register that is initialized with the allowable time for each routine and is counted down by the computer clock. When the register that contains the timer quantity shows a negative value, this indicates that the allowable execution time has been exceeded, or, in short, that a "time-out" has occurred. Because the watchdog timer is used to monitor all application routines, the

instructions for implementing this function are part of the scheduler. The allowable time is, of course, specific for each routine.

Finally, to complete the basic structure, it is proposed that it include a provision for an abort return. This is to preserve the integrity of the rest of the software in the event all alternates of a given sequence or module fail. The abort return is equivalent to the ERROR declaration in the recovery block (Figure 1). Call for an abort might cause the executive to generate an "essential task list" to substitute for the normal task schedule as a basis for proceeding. A diagnostic routine could be invoked to determine the period of suspension for the failed software module. If it is the first failure for a given module, the suspension may be lifted immediately upon a new call to the program. On the other hand, if repeated failures have been observed, the module may be suspended until a manual intervention or a change in the flight condition has taken place.

Figure 4 is a schematic representation for the basic structure discussed above.

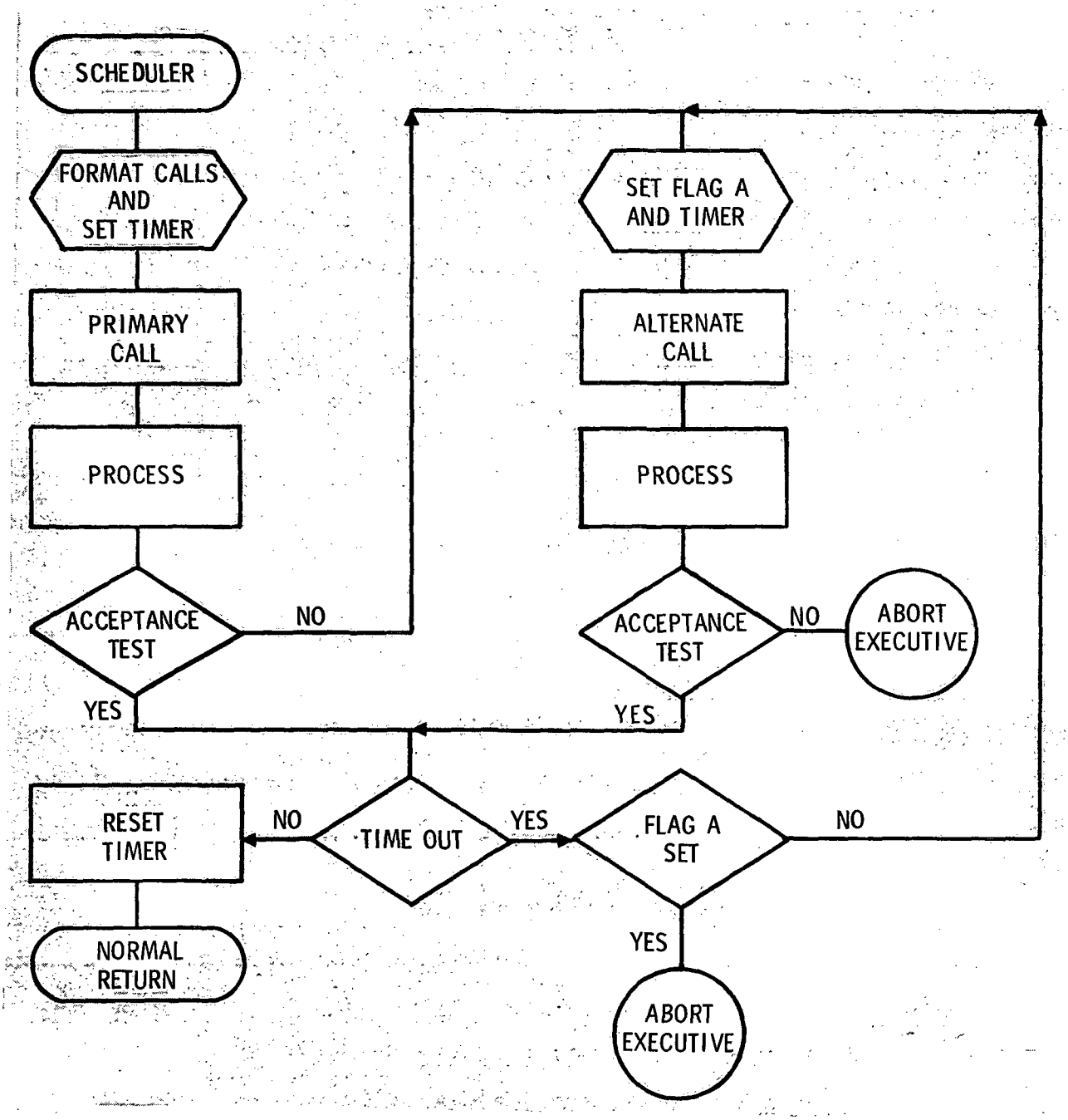


FIG. 4 FAULT TOLERANCE FOR APPLICATION MODULES

Exemplary Implementation

As an example of the implementation of fault-tolerant software, we take an East/West position routine in an aircraft navigator. This routine computes present longitude of the aircraft for display to the pilot and for automatic guidance of the aircraft (by comparison with a desired position determined from the flight plan).

The East/West position program is one of a number of applications that can be called by the scheduler. The scheduler formats two calls for each application (to the primary and to the alternate routine), and these preferably involve different calling parameters. The primary E/W position is computed from the output of an inertially stabilized accelerometer, and the calling parameter for the primary module is the E/W velocity (E_VEL) read by the accelerometer (an integrating accelerometer and a fixed time interval between readings is assumed). The backup module computes longitude from airspeed and compass heading, and the calling parameters are true airspeed (TAS), true heading (HDG), and a wind speed correction (E_WIND) that was computed by another module from prior inertial information. After the calling data has been loaded into the appropriate memory locations, the watchdog timer is set, and the transfer to the primary module is made. The process of the primary module is essentially

NEW_LONG = PREV_LONG + (E_VEL/SCALE_FACT)

where the scale factor is a function of the accelerometer calibration and of latitude.

NEW_LONG computed by the primary module is then subjected to an acceptance test that includes a typical threshold criterion:

$$NEW_LONG \geq PREV_LONG + (PREV_LONG - NEXT_PREV_LONG) - K$$

and

$$NEW_LONG \leq PREV_LONG + (PREV_LONG - NEXT_PREV_LONG) + K$$

where K represents the threshold for the test. If this test is passed, NEW_LONG is returned to the executive, and the timer is reset.

If the test is not passed, or a time-out is experienced, a call to the backup module is immediately issued, and the previously stored backup parameters are then utilized. Also, flag A is set and the timer is reset. The backup module computes

GRNDSP = E_WIND + TAS * SIN (HDG)

NEW_LONG = PREV_LONG + (GRNDSP/SCALE_FACT2)

where a new scale factor is utilized. The resulting NEW_LONG is subject to the previously stated acceptance test, although in some applications a different value of K might be used to allow for discontinuities due to changing from one program to another. If the acceptance test is passed, NEW_LONG is passed to the executive and inserted in the data base.

The remainder of the navigation program then executes in a manner completely independent of difficulties encountered in the E/W position routine. It is indeed essential for a manageable control structure that one alternate program path is not dependent on execution of alternate program paths for any other application routines.

For monitoring purposes, the fact that the backup module has been invoked is visible in the state of flag A. This flag is also essential to prevent continued looping in case a time-out is incurred during execution of the backup module.

Independent Design for Alternate Components

To gain maximum benefit from the redundancy inherent in the recovery block concept it is desirable that the primary and the alternate routines be as independent of each other as possible. Aircraft control systems offer many opportunities for computing control inputs for a given function in several ways, e.g., the yaw steering command may be computed from a rate gyro signal or from a compass signal. This permits independent specifications to be written for the two (or more) routines that are utilized in a recovery block, provides independent data sources, and greatly reduces the possibility that both routines will fail at the same time.

Acceptance Tests

The acceptance test, which is an essential feature of the recovery block, can for many aircraft control functions be rather simply implemented by comparing present and previous values of the computed quantity. Previous values would be automatically available in the "recursive data cache". The laws of physical continuity require that the difference between successive results be small, and when the difference exceeds a specified level this indicates failure.

Current avionics computers frequently have hardware traps for continuous monitoring of overflow or underflow, use of illegal operation codes, and accessing unauthorized memory areas. All of these provisions can be organized to detect deviation of a computer program from expected performance. They are currently utilized to halt or abort processsing, but they can obviously be used as acceptance test inputs in the fault-tolerant software structure. Hardware provisions for those tests are therefore identified as computer architecture features that enhance the capabilities of software fault tolerance.

Many aircraft control functions require that results of a computation not only be correct, but that they be supplied in a timely manner so as to meet the accuracy and stability

criteria of the control system. For this reason it is essential to include, as an acceptance test, a test for timing of the program execution. In the fault-tolerant software structure proposed here this is accomplished by a watchdog timer, a special register* that is initialized with the allowable time for each routine and is counted down by the computer clock. Provision for a watchdog timer register, preferably with hardware real-time clock decrement, is another architecture feature that is desirable for software fault tolerance.

Figure 5 shows more detail of a primary application module. This expansion is intended to illustrate a number of optional acceptance test implementations, consisting of a series of separate tests for correctness of the call procedure, input operations, and processing. Overflow, underflow, and other hardware-implemented tests may also be incorporated. Although the actual test structure is more complex than that shown in Figure 4, it has a single YES and a single NO exit and is a logical replacement for the simpler structure.

---

*The watchdog timer function can also be implemented in software, in which case a memory location is used instead of the register. The execution time penalty may make this a less desirable alternative.

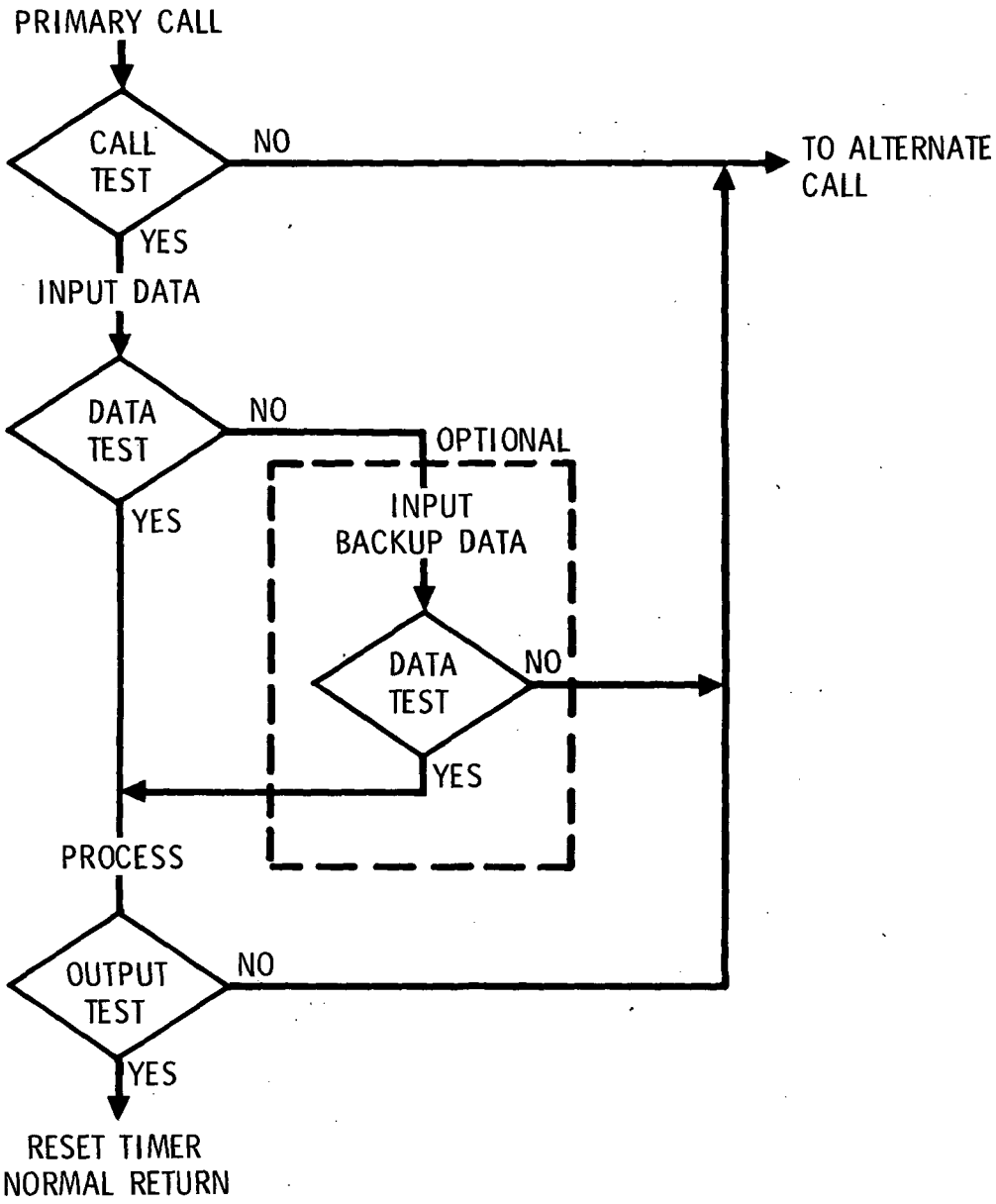


FIG. 5 INTERNAL STRUCTURE FOR PRIMARY APPLICATION MODULE

The reason for the separation of the test blocks is that certain error-prone operations (primarily transfers and data inputs) should be checked at the earliest possible time, both because the test can be more sensitive at that point and to prevent contamination of other program quantities. These tests sometimes are incorporated in the current practices for very critical software projects and are termed "defensive programming". The significant feature of the fault-tolerant software concept is that the error exit from all tests goes to the alternate call, thus resulting in a simple and uniformly applicable control structure.

A variant of this general rule is shown in the optional backup data sequence in Figure 5. Use of this structure may be desirable where the primary data input is undependable or intermittent, and where a source of backup data is readily available (e.g., primary input data from the previous cycle). It should be observed that, here again, the primary data test and backup data test can be collapsed into a structure having a single YES and a single NO exit. The resulting structure corresponds to a recovery block within a recovery block, as illustrated in Figure 2. If the backup data sequence is not to be used, the NO exit from the first data test will continue directly to the alternate call.

The purpose of the call test is to determine that the correct module has been reached and that calling parameters

have been passed correctly. A suitable implementation is that the executive, in formatting the call, creates a checksum over the called address and the calling parameters. In the called module itself are stored (in a different memory location) its starting address and the locations where the callng parameters are expected. Checksumming over these locations, and comparison with the checksum received as part of the call, concludes the call test. Where the computer hardware provides extensive error-detection capability for memory access and readout, the call test may not be required.

The data test can address correctness of transmission, correctness of content, or both. To determine correctness of transmission any one of a number of error-detecting codes can be utilized, and some of these can be retained as an aid to fault diagnosis (primarily hardware-oriented) in further processing. Checksums over blocks of data are, of course, also possible. To determine correctness of data content, the data type can be checked, and increment tests can be performed.

The output test can employ the correctness features mentioned above, or explore correlation between input and output data. Where the application module serves as part of a closed-loop control system, the special properties of such systems (e.g., expected smoothness of control) should be incorporated in the output test. Selective verification of program assertions may also be employed in this test

(Ref. 6). If direct output to aircraft controls is to be furnished by an application routine, then this should be accomplished after successfully passing the output test and before returning to the executive.

Degraded Recovery Block

The additional resources, both for software development and computing hardware, to implement all functions in the recovery block structure may not be warranted. In practice, applications software is likely to consist of a mix of single-string modules and recovery block modules. However, to prevent hang-up of the entire computer due to faults in the single-string modules, some partial fault tolerance provisions are required even in the single-string modules. Thus, a "degraded" recovery block structure without backup alternate routines is defined for those functions that are not so critical as to warrant the redundancy provided by the full recovery block. The degraded recovery block would employ the watchdog timer and monitoring of hardware error flags as acceptance test criteria. In the event of failure an abort return would be called. In this way, results derived from, for example, overflow conditions can be prevented from entering the system, as well as infinite looping or other failures that would absorb excessive computer time.

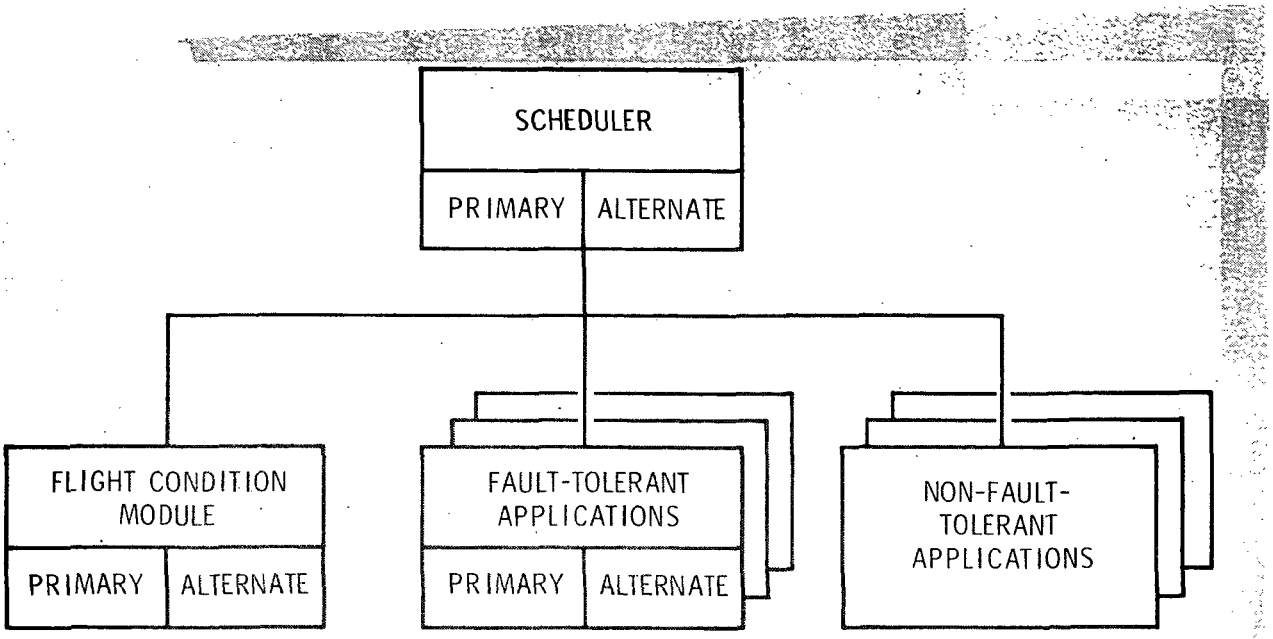
Scheduler and Flight Condition Module

The overall design of the fault-tolerant software

must also provide for a scheduler and for a flight-condition module. The scheduler ensures that tasks are accessed in a proper sequence and that all tasks assigned to a specific time period are properly executed. The requirement to access certain tasks depends on flight condition (e.g., automatic landing tasks need not be serviced during the cruise mode) and, therefore, a module that determines flight condition must also be provided. Flight condition is also an essential input in determining acceptance test criteria such as allowable surface deflection, and limits on rates of climb and descent, etc. Both the scheduler and the flight-condition module are essential for the operation of the overall software system and must therefore be treated as flight-critical functions and coded as recovery blocks.

Fault-Tolerant Aircraft Control Software

From these considerations an overall structure for fault-tolerant aircraft control software emerges that is shown in Figure 6. The scheduler, flight-condition module, and those flight-critical (criticality category 1) applications listed in Table 1, Section 4, will need to be structured as recovery blocks with at least one alternate program. Software for all other functions can be structured as degraded recovery blocks with just a primary program (the originally intended one) and minimal acceptance tests that detect overrun of the watchdog timer or violation of some of the computer hardware-monitored



FUNCTIONS

- Attitude Control and Stability Augmentation
- Flutter Control
- Automatic Landing
- Attitude Director Indicator
- System Monitoring Management*
- Life Support Monitoring Management*
- Engine Systems Control*

ALL OTHER FUNCTIONS
(See Table 1, Section 4)

*Implemented in a fault-tolerant structure if determined to be flight-critical (See Table 1, Section 4, and Reference 2)

FIG. 6 STRUCTURE FOR FAULT-TOLERANT AIRCRAFT CONTROL SOFTWARE

constraints. Upon failure of the acceptance test the program is simply not executed and a warning message is displayed to the crew.

Critical Function Backup

Conceptually, an additional reliability improvement can be obtained by backing up the fault-tolerant computer and software for the critical aircraft control functions with a separate simplex computer executing the simplest possible coding of the required software. A block diagram of such an arrangement is shown in Figure 7. Switching to the simplex computer brings entirely different software into play and removes any unexpected hardware-software interactions that may have contributed to failure of the fault-tolerant software. Because the entire computing environment is being changed the probability of correlated failures is very small.

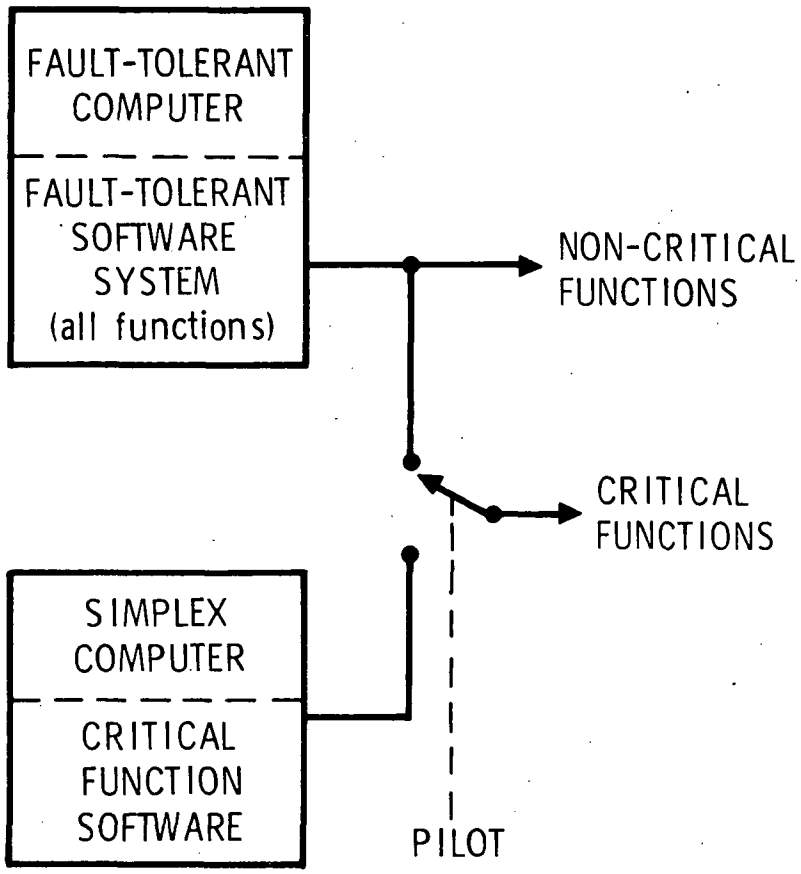


FIG. 7 CRITICAL FUNCTION BACKUP

Section 4

RESOURCE REQUIREMENTS FOR FAULT-TOLERANT SOFTWARE

The fault-tolerance provisions described in the preceding section involve the expenditure of additional resources over and above those normally required for both the software development and the operation of an airborne computer. The most obvious penalty compared to conventional software is that two routines have to be coded and stored where previously only a single one was required. Further, an acceptance test has to be added for each recovery block, and this involves additional code that has to be developed, stored, and executed. If the entire aircraft control software were to be coded as recovery blocks the development budget would have to be more than doubled. Moreover, memory requirements for the computer on which this software is to execute would be increased by a like factor, and execution time would be increased due to the running of the acceptance tests every time a recovery module is exited. Since the computers on which this software is expected to execute employ redundant implementation of memory and processors, it was assumed that gross expansion of the memory and processing requirements to accommodate the entire software in the form of recovery blocks would not be acceptable. Therefore, it was decided to restrict the recovery block format to those modules that are assumed to be critical to safe flight and landing of the aircraft. The

classification of aircraft control software by function, criticality, and computer resource requirements as prepared by Stanford Research Institute (Ref. 2) was used as a reference to determine which functions to implement in this manner.

Essential data from the SRI reference (contained in their Tables 2 and 3) are, with minor editing, reproduced in Table 1. An explanation of the criticality levels assigned by Stanford Research Institute (SRI) is shown in Table 2. Only functions at criticality level 1 or 2 were considered for implementation in the full recovery block treatment outlined in the preceding section. Moreover, some deletions seem possible even from that restricted set of functions. For example, the inertial navigation function (No. 8) has functional backup through Functions 9 or 10, except for over-ocean flight. Furthermore, assuming successful development of the NAVSTAR Global Positioning System, it seems that complete backup for inertial navigation will be available in the post-1980 time frame. Under these assumptions, complete fault-tolerant software treatment for this function might not be necessary. Because of the uncertainty in the criticality of several of the support systems it was concluded that it would be premature to make a firm decision on their treatment with regard to fault-tolerant software. For the engine control system a major portion of the code is believed to be concerned with economy of operation and only a rather minor part is truly critical

TABLE 1

ANALYSIS OF AIRCRAFT CONTROL COMPUTATIONAL REQUIREMENTS

Function No.	Description	Criticality	Instruction Storage Reqt. (Words)	Timing Reqt (Relative Ops/Sec)
Attitude and Flight Path Control				
1	Digital attitude control (with stability augmentation)	1	1845	9.9
2	Active flutter control	1	70	27.6
3	Active gust maneuver load control	3 or 5	45	5.6
4	Automatic landing	1	750	22.0
5	Other flight path control	4	150	-
6	Electronic attitude director indicator	1	790	30.8
Area Navigation and Related Functions				
7	Supervisor (mode selection, etc.)	4	75	-
8	Inertial	2	2100	13.5
9	VOR/DME	4	250	1.7
10	Alternate radio navigation (multiple DME, omega)	4	400	-
11	Air data	4	110	
12	Optimal combination (Kalman filter)	4	250	.2
13	Processed flight data (enroute)	4	450	11.0
14	Airspeed, altitude	4	360	3.6
15	Graphics display	4	890	12.8
16	Text display	4	640	7.6
17	Automatic tuning, flight-leg switching, etc.	4	200	-
Communication, ATC				
18	Collision avoidance	4	550	8.5
19	Aircraft (internal) data communication	--	280	2.8
20	Air/ground/air data communication	4	550	0.3

TABLE 1

ANALYSIS OF AIRCRAFT CONTROL COMPUTATIONAL REQUIREMENTS (continued)

Function No.	Description	Criticality	Instruction Storage Reqt. (Words)	Timing Reqt (Relative Ops/Sec)
Support Systems				
21	Aircraft integrated data system	5	650	0.6
22	Instrument monitoring	4	1800	5.6
23	System monitoring and management	1 to 4	900	0.5
24	Life support system monitoring and management	1 to 4	900	0.5
25	Engine systems control and operation	1 to 2	1300	47.5
TOTAL			16305	212.5

*Excerpted from Tables 2 and 3 in Reference 2

TABLE 2

EXPLANATION OF CRITICALITY LEVELS

Criticality Level	Effect of Failure	Example
1	Immediately affects safety of flight	Stability augmentation for inherently unstable aircraft
2	May affect safety of flight at a future time	Altitude or airspeed display
3	Change in mission required to avoid safety degradation	Gust alleviation failure requiring airspeed limitation
4	Imposes substantial operational penalties	Navigation or communication failure
5	Produces economic loss only	Failure of engine trim control

to safety; therefore, this system too has been deleted from the critical functions as far as estimating the requirements for fault-tolerant software is concerned.

To estimate the storage and timing requirements for the fault-tolerant software, it was assumed that the backup program in each recovery block would have the same number of instructions as the primary program identified by SRI and that acceptance tests will require 15 percent of the instructions of a primary program. It was also assumed that all less critical applications would be coded as "degraded" recovery blocks. In addition to the regular code this will require three instructions for setting up the watchdog timer and monitoring of hardware error flags. Instructions required by acceptance tests will have to be executed every time a module is called, but instructions for backup programs will not ordinarily have to be executed and thus will not affect the timing requirement.

As shown in Table 3, it is estimated that the use of full recovery blocks for the reduced set of functions and of degraded recovery blocks for the remaining functions will require an increase in the storage requirements of approximately 30 percent, but the impact on program execution speed is estimated to be negligible. These estimates may be unduly pessimistic because some of the reasonableness tests incorporated in the formal acceptance test of a recovery block

TABLE 3

IMPACT OF FAULT-TOLERANT PROVISIONS ON COMPUTATION REQUIREMENTS

Function No.	Description*	Criticality*	Storage Requirement (Words)		Timing Requirement (Relative Ops/Sec)	
			Original*	Fault Tol	Original*	Fault Tol
Attitude and Flight Path Control						
1	Digital attidue control (with stability augmentation)	1	1845	4251	9.9	12.0
2	Active flutter control	1	70	168	27.6	32.0
3	Active gust maneuver load control	3 or 5	45	48	5.6	5.6
4	Automatic landing	1	750	1732	22.0	26.0
5	Other flight path control	4	150	153	-	-
6	Electronic attitude director indicator	1	790	1824	30.8	36.0
Area Navigation and Related Functions						
7	Supervisor (mode selection, etc.)	4	75	78	-	-
8	Inertial	2	2100	2103	13.5	13.5
9	VOR/DME	4	250	253	1.7	1.7
10	Alternate radio navigation (multiple DME, omega)	4	400	403	-	-
11	Air data	4	110	113	-	-
12	Optimal combination (Kalman filter)	4	250	253	.2	.2
13	Processed flight data (enroute)	4	450	453	11.0	11.0
14	Airspeed, altitude	4	360	363	3.6	3.6
15	Graphics display	4	890	893	12.8	12.8
16	Text display	4	640	643	7.6	7.6
17	Automatic tuning, flight-leg switching, etc.	4	200	203	-	-
Communication, ATC						
18	Collision avoidance	4	550	553	8.5	8.5
19	Aircraft (internal) data communication	-	280	283	2.8	2.8
20	Air/ground/air data communication	4	550	553	0.3	0.3

TABLE 3

IMPACT OF FAULT TOLERANT PROVISIONS ON COMPUTATION REQUIREMENTS (continued)

Function No.	Description*	Criticality*	Storage Requirement (Words)		Timing Requirement (Relative Ops/Sec)	
			Original*	Fault Tol	Original*	Fault Tol
Support Systems						
21	Aircraft integrated data system	5	650	653	0.6	0.6
22	Instrument monitoring	4	1800	1803	5.6	0.6
23	System monitoring and management	1-4	900	903	0.5	0.5
24	Life support system monitoring and management	1-4	900	903	0.5	0.5
25	Engine systems control and operation	1-2	1300	1303	47.5	47.5
TOTAL			16305	20888	212.5	228.2

*See Tables 2 and 3 in Reference 2

may be included anyway as a defensive programming technique for any critical software program. Also, because the alternate program segments of the recovery block frequently make use of sensor data that have to be processed in any case, the code length for these may not in all cases be quite as long as the primary segment.

In addition to the application functions defined in Tables 1 and 3, the overall design of the fault-tolerant software must also provide for a scheduler and for a flight-condition module. The impact of these additional functions on storage and timing requirements should be quite small. It is estimated that the scheduler and flight-condition module together may comprise 200 primary instructions; the iteration rate for the scheduler is assumed to be five per second, and that for the flight-condition module one per second.

Section 5

A FAULT-TOLERANT SCHEDULER AND FLIGHT CONDITION MODULE

The scheduler is a very essential component of the overall fault-tolerant software system. The scheduler ensures that tasks are accessed in a proper sequence and that all tasks assigned to a specific time period are properly executed. The requirement to access certain tasks depends on flight condition (e.g., automatic landing tasks need not be serviced during the cruise mode) and, therefore, a module that determines flight condition must also be provided. Flight condition is also an essential input for acceptance tests such as determining allowable surface deflection, and limits on rates of climb and descent, etc. Both the scheduler and the flight-condition module are essential for the operation of the overall software system and must therefore be treated as critical items and coded as recovery blocks.

For some tasks, all data required for the computation may be available within the computer or can be accessed from peripherals whenever necessary. These tasks can always be executed in a fixed order. There are, however, other applications in which instruments furnish outputs at irregular frequencies or in which communications are necessarily asynchronous, e.g., the ground communications task (Functions 19 and 20 in Table 1). To service these applications it is necessary that a data buffer be set up (either in the computer

memory or in the peripheral device), and that a "data-ready flag" (a logical quantity) be set "true" whenever new data are furnished and set "false" whenever the data have been read.

Also, because the time taken to complete certain tasks (particularly those involving asynchronous communications) cannot always be predicted, it may be necessary to assign priorities to certain routines in order to make sure that they are carried out at least as often as is required to maintain safe control of the aircraft. Thus, to be able to service the general spectrum of aircraft control tasks the scheduler must be capable of coping with asynchronous data transfer and to recognize various task priorities.

The design of the primary routine for the fault-tolerant scheduler was therefore governed by the following requirements:

1. There must exist the ability to transfer data from peripheral devices without action of the scheduler or of application modules.
2. The transfer of such data will include the setting of a ready flag and accessing by the computer of these data will be accompanied by resetting of the ready flag.
3. No process will be interrupted during its operation.
4. The scheduler will give control to the highest priority process that has a current ready flag.

The principal fault-tolerance provisions that must be incorporated in the primary scheduler routine are

1. It must detect if an application module does not return within a prescribed interval (the watchdog timer function described in Section 3), and
2. It must recognize when it hangs up (it must perform its own acceptance test).

The key to the scheduler design is to organize aircraft control tasks on a cyclical basis, with the major cycle being perhaps on the order of 0.5 to 1 second. The major cycle is identified as the interval during which each task should be serviced at least once. The end of a major cycle is identified by a clock pulse derived from computer hardware (and itself considered fault tolerant in the present context). If the scheduler has operated correctly each process should then have been accessed at least once, and this action can be made visible by incorporating "serviced flags" at the beginning of each task. Right after the occurrence of the major clock pulse these flags are interrogated, and the appearance of a non-serviced state indicates a fault in the primary scheduler routine. If they are all in the serviced state, however, they are reset to the non-serviced state, and the new major cycle servicing begins.

The requirements discussed so far have been incorporated in the primary scheduler design shown in Figure 8.

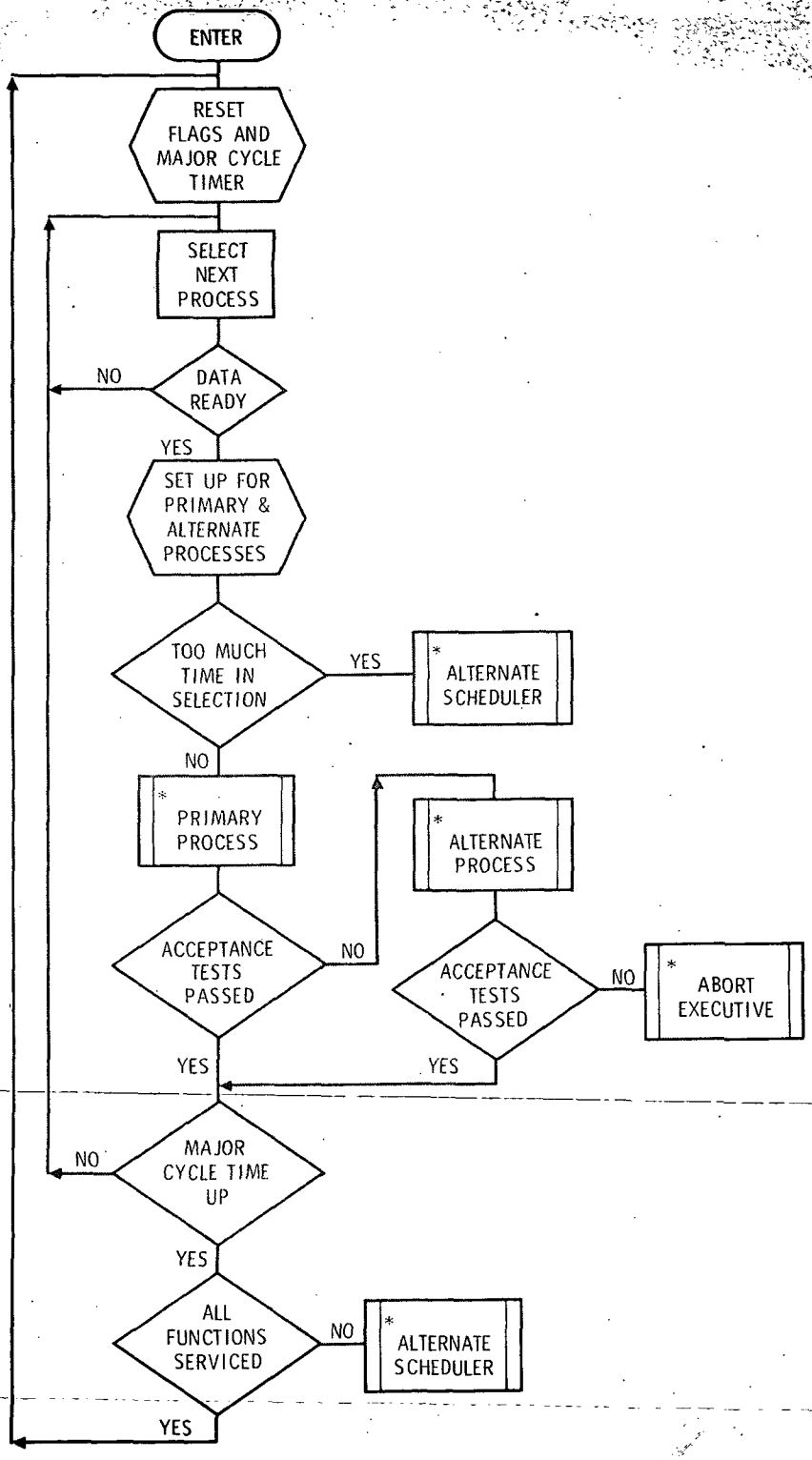


FIG. 8 PRIMARY SCHEDULER

Because the most likely place at which the scheduler could malfunction is in the selection of the next task, a separate test based on time in selection has also been incorporated in the design. If too much time is spent in selection, the alternate scheduler is accessed. This feature is optional. A PASCAL version of this scheduler is contained in Appendix 2.

The functional implementation for the backup scheduler is deliberately kept as different as possible from the primary one. An example of how this can be accomplished is shown in Table 4, where the implementation of the scheduler functions in the primary and backup schedulers are compared. A flow chart of the alternate scheduler that embodies these functions is shown in Figure 9.

When the backup scheduler is invoked it is intended that servicing of all non-critical functions be deferred. Because all functions then remaining are critical, it is not necessary to have a priority structure and a fixed order of service can be implemented. The processes identified as criticality level 1 or 2 in Table 1 do not depend on asynchronous data transfer and, for this reason, it is also believed that the data-ready test can be avoided in the backup scheduler. At the end of each major loop the primary scheduler checks for servicing of all processes in order to make a decision whether to resume its own scheduling or to transfer

TABLE 4

COMPARISON OF PRIMARY AND BACKUP
SCHEDULER IMPLEMENTATION

Function	Primary	Backup
Processes scheduled	All	Critical only
Order of service	Priority with data-ready test	Fixed
End of major loop	Checks for service of all processes	Transfer to its start
Transfer to other	For test only	Periodically

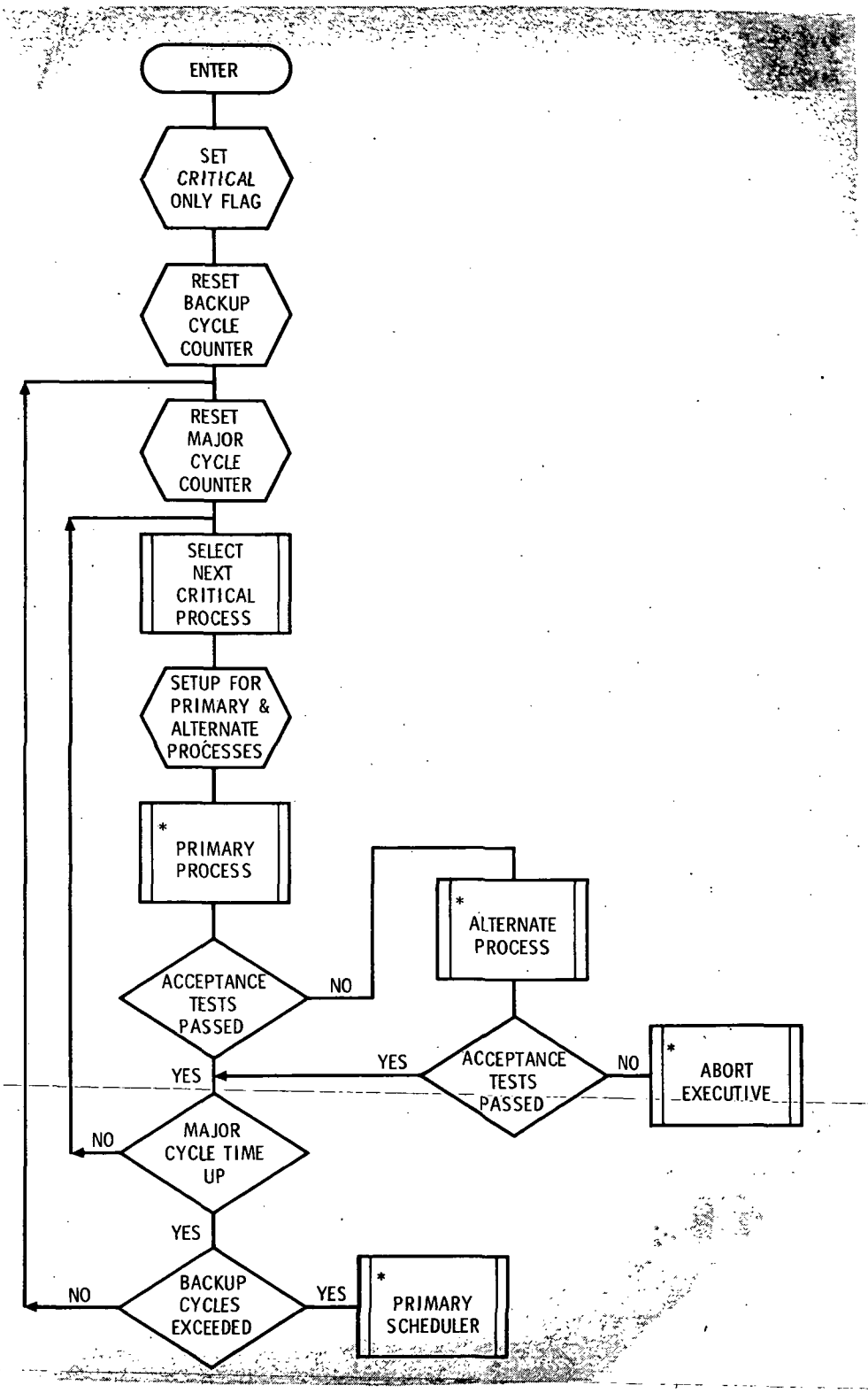


FIG. 9 ALTERNATE SCHEDULER

to the alternate scheduler. Such a decision is not appropriate for the backup routine, and it simply transfers to its own start at the end of every major cycle.

Because the primary scheduler represents thoroughly checked out software that has been performing adequately until just prior to the transfer to the backup, it is presumed that the transfer was most probably due to the occurrence of an unusual event that is not likely to be of a permanent nature. Because the operation of the backup scheduler exposes the aircraft to a less satisfactory environment (servicing of non-critical processes has been suspended) it is desirable to attempt periodic transfers back to the primary scheduler to restore normal operation. Thus, the backup scheduler will periodically transfer to the primary module, and only upon unsatisfactory operation of a major cycle on the primary scheduler will the operation of the backup scheduler then be resumed. The suspension of service to non-critical functions will have to be signaled to the crew. It may also be desirable to permit manual switching to the backup executive in case of unsatisfactory operation of the aircraft control system that goes somehow undetected by the acceptance tests.

The total fault-tolerant scheduling system then consists of the primary and the alternate scheduler connected as shown in Figure 10. Operation commences in the primary

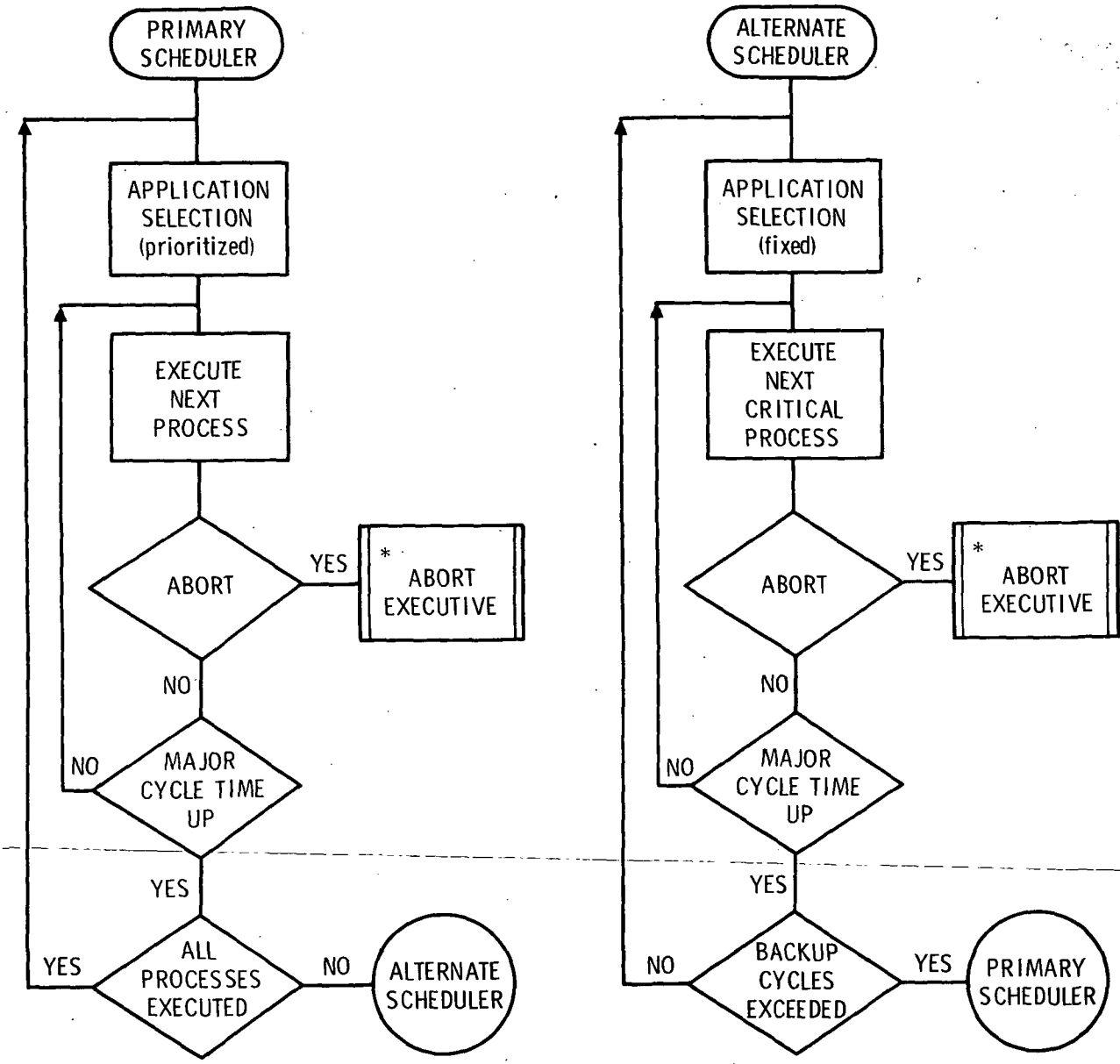


FIG. 10 FAULT-TOLERANT SCHEDULERS

scheduler mode and it will remain in that mode unless one of the following two events occurs: an individual process does not execute, or all processes are not executed within a major cycle time.

That an individual process does not execute means the routine did not pass the acceptance test within the allowed time. This event results in the abort return indicated in Figure 4. The event that all processes have not been executed within a major cycle has already been discussed as the principal acceptance test for the operation of the primary scheduler. If this test is not passed there is, therefore, a need to invoke the alternate scheduler.

The operation of the alternate scheduler follows that described previously until the predetermined number of backup cycles are exceeded, and then automatic switchback to the primary scheduler will take place.

The fault-tolerant scheduler outlined here provides protection against hangup due to failure in any application module (through the watchdog timer), hangup due to failures in the primary scheduler (by monitoring for execution of all processes at major cycle time), and against correlated failures of primary and alternate schedulers by an independent specification and implementation technique. The system was

described here in operation against a priority-oriented task list. It is, however, quite flexible, and its operation against a major-minor-cycle-oriented task list will now be discussed with reference to Figure 11.

The frequency with which the various automatic control processes aboard an aircraft need to be serviced by the computer varies considerably. For this reason a common procedure for structuring the servicing is to divide the processes into a hierarchy of major and minor timing cycles, as illustrated in Figure 11, where a major cycle is divided into two intermediate cycles which, in turn, are divided into two minor cycle segments. The number of layers (here major, intermediate, minor) and the number of lower layer cycles represented by an upper layer (here given as two) can, of course, vary quite a bit. All tasks need to be serviced once during the cycle to which they are assigned, i.e., once per major, intermediate, or minor cycle. One way of accomplishing this is by the allocation shown in Figure 11. At the beginning of each minor cycle, e.g., between events 0 and 1, 2 and 3, 5 and 6, and 7 and 8, the minor cycle tasks are serviced. In the remainder of the minor cycle time, e.g., between events 1 and 2, 3 and 4, 6 and 7, and 8 and 9, the intermediate cycle tasks are accomplished, and if any time is left over in the minor cycle, e.g., between events 4 and 5, and 9 and 10, then the major cycle tasks are addressed. The

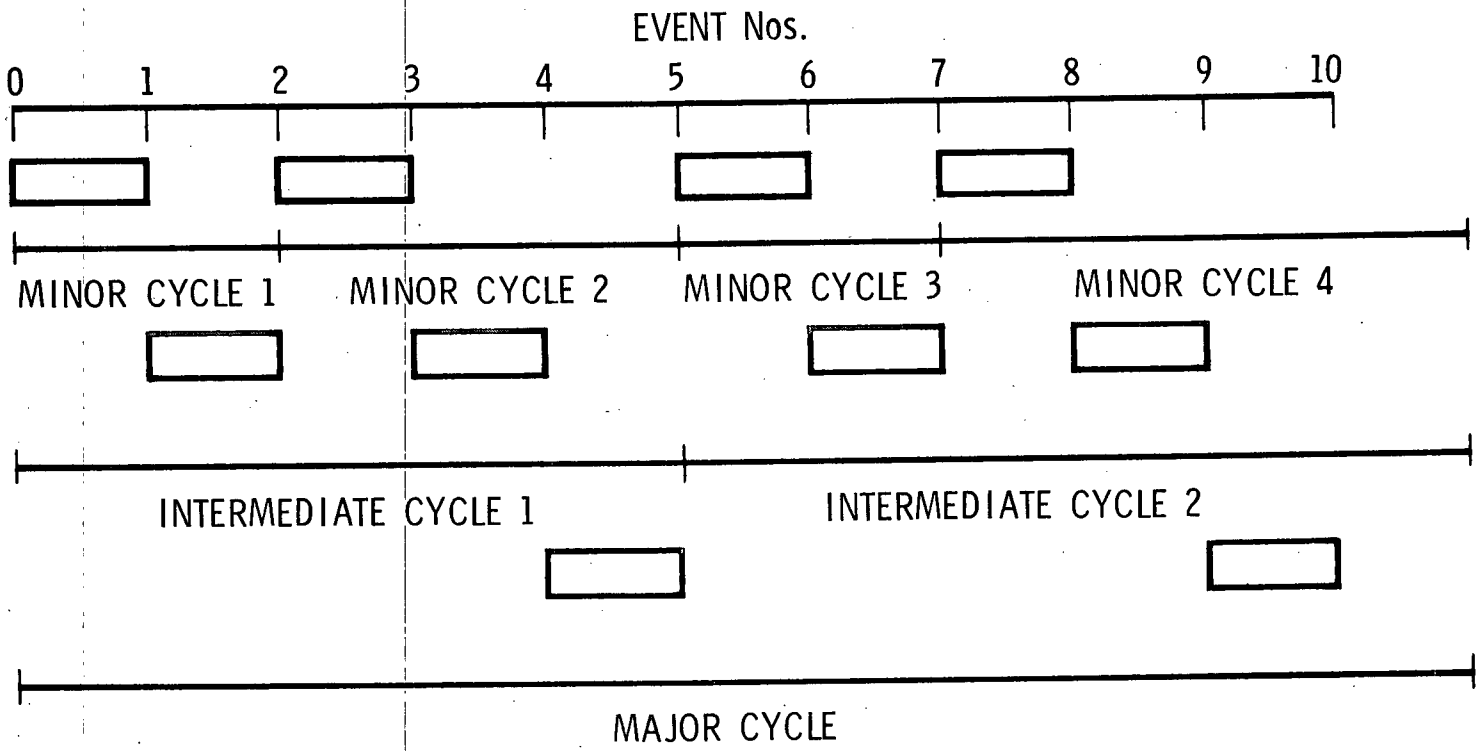


FIG. 11 MAJOR-MINOR CYCLE SCHEDULING

program must be designed so that all major cycle tasks can be accomplished under the worst condition within one major cycle, and the process can be repeated over again. As shown in Figure 11, there is indeed idle time between event 10 and the end of the major cycle.

The scheduling required in Figure 11 can be handled by the primary scheduler in the following manner. All minor cycle tasks are assigned priority 1, intermediate cycle tasks priority 2, and major cycle tasks priority 3, with priority 1 being the highest. Data-ready flags are set "true" for all tasks at the beginning of the major cycle. (The scheduling shown in Figure 11 inherently assumes that tasks can call data.) Data-ready flags for the minor cycle tasks will be set false upon execution of these tasks and can then be set true again at the beginning of each minor cycle, e.g., at events 2, 5, and 7 in Figure 11. Similarly, the data-ready flags for intermediate tasks can be reset at the beginning of the intermediate cycle (e.g., at event 5). At the beginning of each new cycle (minor, intermediate, and major) a signal transition is assumed to occur in the computer hardware, and this transition can then be selected to reset the flag registers as outlined above. The scheduling of the same tasks by the backup scheduler would require preparation of a task list that interleaves critical minor, intermediate, and major cycle tasks with, if necessary, some wait times in between

to assure proper operation. The critical tasks should occupy only a fraction of the major cycle, so there is no danger of an overlap situation (insufficient time to finish all required processes).

The flight condition module is a function closely associated with the scheduler. It is assumed that flight conditions are sensed primarily in terms of horizontal and vertical velocities, e.g., cruise condition is identified with having reached a specified airspeed. It is also assumed that there are at least two completely independent sensor systems for horizontal velocity, e.g. inertial and air data, and three for vertical velocity, e.g. inertial, air data, and radar; and that each sensing system is itself sufficiently reliable (or redundant) that the sensor systems will furnish a valid input to the flight condition module through algorithms not specifically considered here.

Because several significant uses of flight condition information involve aerodynamic effects (e.g., limitations on control surface travel), it appeared natural to select true airspeed (an output of the air data computer) as the input to the primary routine and velocity information from the inertial navigation system as an input to the alternate routine. For operation in proximity to the ground (i.e., initial climb, descent, and ground operation), altitude and rate-of-climb information derived from radar altimeter data are used as

inputs to the primary routine, and are derived from air data signals for inputs to the alternate routine.

An example of the resulting signal selection is shown in Table 5. Alphanumeric symbols represent velocity and altitude points for transition from one flight condition to another. Numerical values for these transitions can be tailored to the specific mission (e.g., takeoff weight, runway altitude, and temperature at origin or destination).

TABLE 5

SENSOR INTERACTION FOR FLIGHT CONDITION MODULE

Condition Sensed	Primary Flight Condition Routine		Alternate Flight Condition Routine	
	Input Data Used	Acceptance Test	Input Data Used	Acceptance Test
Ground	Radar altitude < a1	Landing gear oleos compressed	Air data computer true air speed < tas1	Inertial navigator horizontal speed ≤ hs1
Climb	(Radar vertical speed > 0) ∧ (throttle ≥ climbpower)	(Landing gear oleos not compressed) ∧ (radar altitude < a2)	Air data computer true air speed between tas1 and tas2	Inertial navigator horizontal speed between hs1 and hs2
Cruise	Air data computer true air speed > tas2	(Air data computer rate of climb = 0 ± margin) ∧ (throttle < climbpower)	Inertial navigator horizontal speed > hs2	(Inertial navigator vertical speed 0 ± margin) ∧ (throttle < climbpower)
Descent	(Radar vertical speed < 0) ∧ (throttle < descentpower)	Air data computer true air speed < tas2	Air data computer vertical air speed < vs1	(Inertial navigator horizontal speed between hs3 and hs4) ∧ (flaps = down)

NOTE: The symbol ∧ represents logical "and" (both conditions must be true).

Section 6

RELIABILITY OF FAULT TOLERANT SOFTWARE

The purpose of fault-tolerant software is to enhance the reliability over that obtainable from thoroughly checked-out single-string software. This section addresses modeling of fault-tolerant software such that the reliability improvement can be conceptually demonstrated and, given suitable input parameters, numerically assessed. It would be desirable to supply parameters for this model and then to use it to guide the development of the fault-tolerant software. There are at present no authoritative estimates of the reliability of real-time operational software and none at all for the residual failure probability after fault tolerance (the probability of undetected or correlated failures). A parametric approach is therefore taken in which a wide range of software failure probabilities and of the residual failure probabilities is input to the model.

The transition model shown in Figure 12 has been devised to be representative of the failure processes that can be expected in typical real-time software applications. Starting at state 1, when the primary software module executes correctly, the allowed transitions during some arbitrary interval* are the following:

---

* For the numerical results to be derived from this diagram to be valid, it is required that the probability of more than one transition to state 4 during the selected time interval (failure) be negligibly small.

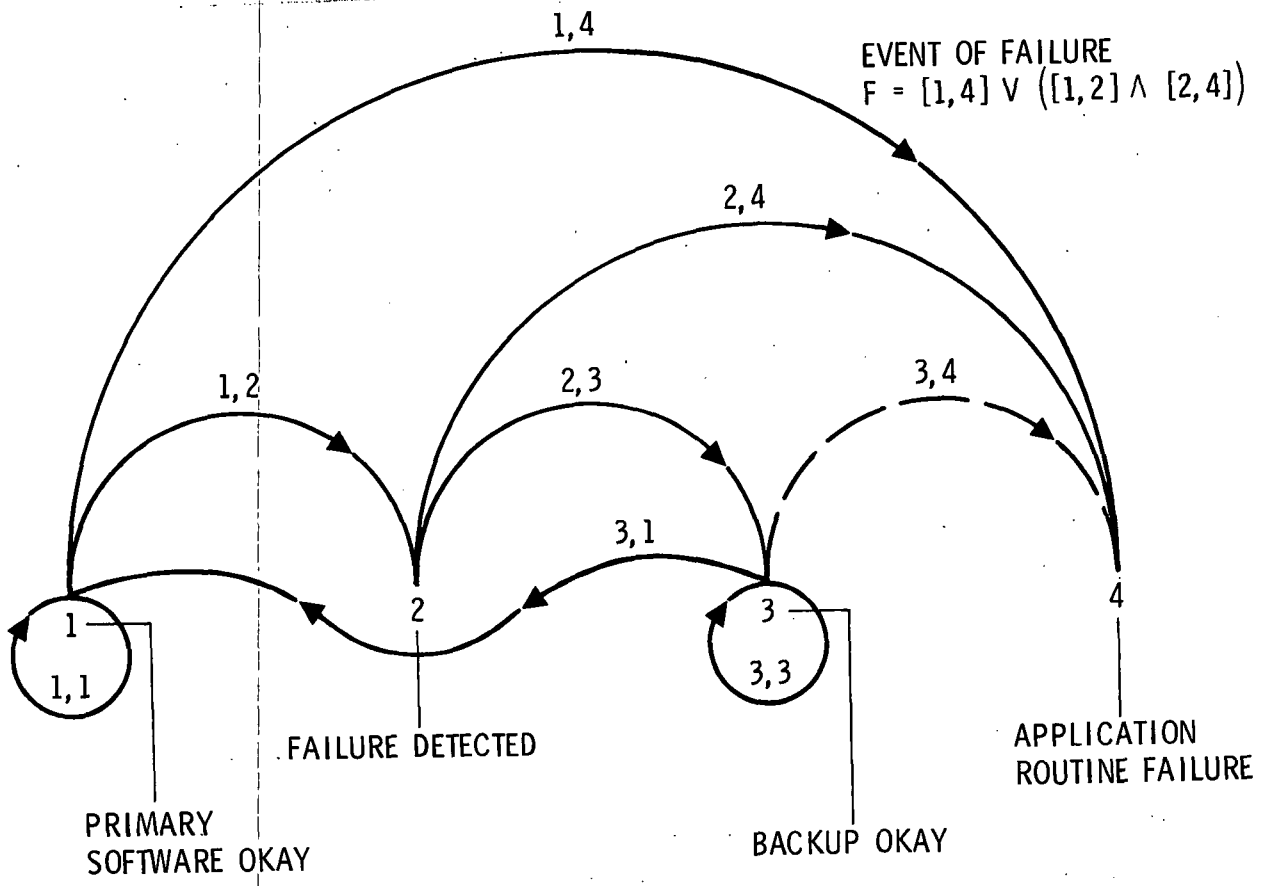


FIG. 12 TRANSITION MODEL FOR AN APPLICATION ROUTINE

Primary software continues to execute correctly (path 1,1).

Primary software fails and the failure is detected (path 1,2).

Primary software fails and the failure is not detected (path 1,4).

In the first case, the process will definitely not cause system failure during the selected interval, and in the third case it definitely will cause system failure. For the case when failure is detected, it has been found convenient to show transition to a pseudostate 2, from which an immediate further transition to either state 3 or state 4 will result. The transition to state 3 represents the case where the backup routine performs satisfactorily for at least one pass of the application program (i.e., the acceptance test is satisfactorily completed). The transition 2,4 represents the case where a failure has been detected, but where the backup routine does not satisfactorily complete the acceptance test. It is assumed that the probability that this will occur due to independent failures in the primary and backup routines is extremely low, but correlated failures, e.g., those due to environmental effects or unforeseen hardware/software interactions, can by no means be ruled out. The transition 2,4 therefore models the event of correlated failure in the primary and backup routines.

If the failure of the primary routine is of a transient nature and if the backup routine has performed satisfactorily, the executive will cause reversion to the primary routine upon the next call to the application module, as modeled by transition 3,1. If the failure of the primary routine is of a more permanent nature, diagnostics may reverse the roles of primary and secondary modules and calls to this application will immediately bring the backup routine into operation. Note that all of this is conditioned on the backup routine having performed satisfactorily at least once after failure of the primary one. Any further failure of the backup routine would therefore be an independent failure. The probability of that event, while the primary routine is still not yielding satisfactory performance, must be considered quite low. For this reason it is assumed that, once the transition 2,3 has taken place (backup routine performed satisfactorily) it will remain in that state or revert back to the primary routine (transition 3,1). The transition 3,4 cannot be completely ruled out, but the probability of this event will be several orders of magnitude less than that of the other transitions once state 3 has been reached. Therefore the arc has been shown dashed, and the probability of this transition is neglected in the following discussion.

In this model the most significant events leading to failure of an application routine are seen to the be the

inability to detect failure of the primary routine (transition 1,4) and, given that primary failure has been detected, the correlated failure of the backup routine (transition 2,4). Subject to previously stated conditions, the probability of failure of an application can therefore be expressed as

$$F_A = P(1,4) + P(1,2) \times P(2,4)$$

A significant deviation from the usual hardware model has thus been established. Probability of failure for redundant hardware elements is modeled by considering failures independent with probability of failure to detect rarely considered in elementary models. For fault-tolerant software, on the other hand, these failure processes (dependency and inability to detect) are seen to be the governing ones. Arbitrarily labeled "residual failure probabilities", they may be represented as an element in series with the independent failure probabilities of the associated application models. However, if our previous assumption is correct that application program failure due to independent failure modes of the primary and backup routines is orders of magnitude less likely than that due to the residual failure modes, then a simplified model consisting entirely of a series arrangement of the residual failure probabilities may be constructed.

Using this model, and arbitrarily assumed values for the residual failure probabilities to predict the corresponding fault-tolerant application program failure probability, a set of figures such as those shown in Table 6 results. These figures are then propagated in Table 7 to a segment of critical software consisting of five programs (e.g., a fault-tolerant scheduler and four fault-tolerant application programs).

Conceptually, an additional reliability improvement can be obtained by backing up the fault-tolerant computer and software for the critical aircraft control functions with a separate simplex computer executing the simplest possible coding of the required software. A block diagram of such an arrangement was shown in Figure 7. Switching to the simplex computer brings entirely different software into play and removes any unexpected hardware-software interactions that may have contributed to failure of the fault-tolerant software. Because the entire computing environment is being changed the probability of correlated failures is very small. However, the probability that software failures will go undetected can no longer be ignored.

Additional research is required to establish the failure probability of representative primary software, and the probability of undetected or correlated failure in a recovery block, in order to make these models useful. In the meantime, the design of fault-tolerant software can proceed on the basis

TABLE 6

FAILURE PROBABILITY OF A FAULT-TOLERANT APPLICATION PROGRAM

Probability of		Primary Failure Probability/Flight			
		10^{-3}	10^{-4}	10^{-5}	10^{-6}
Undetected Error	Correlated Failure	Failure Probability/Flight			
0.05000	0.10000	1.45E-4	1.45E-5	1.45E-6	1.45E-7
0.05000	0.05000	9.75E-5	9.75E-6	9.75E-7	9.75E-8
0.01000	0.02000	2.98E-5	2.98E-6	2.98E-7	2.98E-8
0.00100	0.00100	2.00E-6	2.00E-7	2.00E-8	2.00E-9
0.00010	0.00010	2.00E-7	2.00E-8	2.00E-9	2.00E-10
0.00001	0.00001	2.00E-8	2.00E-9	2.00E-10	2.00E-11

TABLE 7

FAILURE PROBABILITY OF FAULT-TOLERANT SOFTWARE
(FIVE CRITICAL PROGRAMS)

Probability of		Primary Failure Probability/Flight			
		10^{-3}	10^{-4}	10^{-5}	10^{-6}
Undetected Error	Correlated Failure	Failure Probability/Flight			
0.05000	0.10000	7.25E-4	7.25E-5	7.25E-6	7.25E-7
0.05000	0.05000	4.87E-4	4.87E-5	4.87E-6	4.87E-7
0.01000	0.02000	1.49E-4	1.49E-5	1.49E-6	1.49E-7
0.00100	0.00100	9.99E-6	9.99E-7	9.99E-8	9.99E-9
0.00010	0.00010	1.00E-6	1.00E-7	1.00E-8	1.00E-9
0.00001	0.00001	1.00E-7	1.00E-8	1.00E-9	1.00E-10

of taking every possible advantage of multiple sensing of flight parameters and events in order to provide high coverage for the detection of software failures and for reducing the probability of correlated failures.

Section 7

CONCLUSIONS

The recovery block structure as a general implementation of software fault tolerance appears to be suitable to serve specific critical aircraft control functions. The recovery block consists of a primary routine (partial or entire application program), one or more alternate routines, and an acceptance test. If the result of either the primary or of the alternate routine passes the acceptance test, a regular output is delivered to the aircraft control system, and normal operation is maintained. If neither result passes the acceptance test, an abort return is issued that mobilizes other system resources to resolve the difficulty.

Because significant development resources and computer hardware (e.g. memory space and throughput) cost is involved if all application functions were to be implemented in the recovery block structure, the concept of a degraded recovery block has been introduced for non-critical applications. The degraded recovery block employs an acceptance test that includes the watchdog timer, and any error monitoring signals that are provided in the computer hardware. It does not incorporate an alternate routine as in the full recovery block; instead it calls for an abort return to signal the crew to take corrective action while the computer proceeds

with higher priority tasks. The degraded recovery block permits the mixing of unprotected (i.e. without alternates) single-string non-critical software modules with modules using the recovery block structure, by insuring that failure of the non-critical software will not deny access to the essential application modules. It is estimated that flight control software incorporating recovery blocks for critical applications and degraded recovery blocks for all others will only require about 30 percent more storage than non-fault-tolerant software serving the same functions. The execution time penalty for use of fault-tolerant software is estimated to be less than 10 percent. On this basis it is asserted that fault-tolerant software for critical aircraft control functions can be implemented at reasonable expenditure of resources.

The stringent reliability required for real-time aircraft control functions involving safety of flight demand that any potential single failure mode be eliminated. Conventional single-string software servicing critical functions represents a potential single failure mode, and this difficulty can be overcome by the use of fault-tolerant software concepts. The advantages of fault-tolerant software described here, and the rather modest additional resources required for its use, appear ample arguments to proceed with further development and application of this concept.

Since servicing frequency varies widely among the various aircraft control functions and timely execution is essential for all functions in a time-shared multi-task system a fault-tolerant scheduler is an essential and critical component of fault-tolerant software in such a system. The conceptual design of such a scheduler has been described.

An elementary reliability model is described largely to serve as a stimulus and guide for both the acquisition of software reliability data and for the development of a test and validation methodology for fault-tolerant software for the aircraft control application. The need for a considerable data acquisition effort is obvious.

The need for early and intensive development of a test and validation methodology for software to be used as part of a fault-tolerant aircraft control system stems partly from the requirements for demonstrating high reliability of the primary and alternate routines separately, and partly from the need for measurement of the probability of undetected and correlated failures. An approach to planning of these tests needs to be worked out at an early date lest this become the pacing item in the overall effort to develop an aircraft-control concept suitable for energy-efficient transport. The task of demonstrating the achievement of failure rates of the order of 1×10^{-9} per flight hour is not trivial, and perhaps here again fault tolerant hardware techniques can be adapted (Ref. 7).

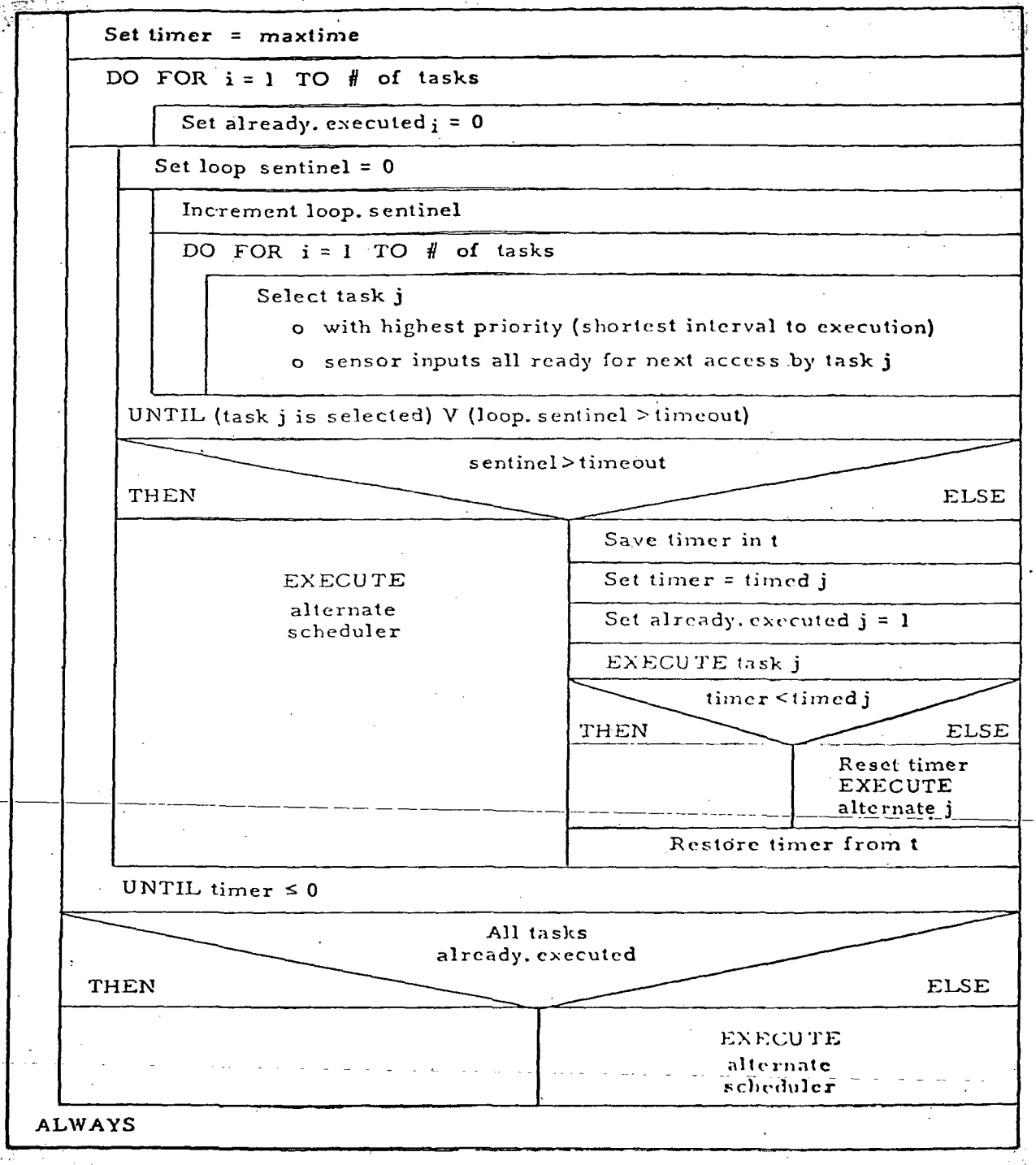
REFERENCES

1. Brian Randell, "System Structure for Software Fault Tolerance," Proceedings 1975 International Conference on Reliable Software, ACM, New York, NY (1975), p. 437.
2. Ratner,R. S.; Shapiro, E. B.; Zeidler, H. M.; Wahlstrom, S. E.; Clark, C. B.; and Goldberg, J.: Design of a Fault-Tolerant Airborne Digital Computer - Vol. II - Computational Requirements and Technology, NASA CR-132253, (1973).
3. Peter R. Kurzhals and Richard Deloach, "Integrity in Flight Control Systems," Proceedings of the AIAA Second Digital Avionics Systems Conference, Los Angeles, CA (1977).
4. Federal Aviation Regulation (FAR) 25, Subpart F, Para. 25.1309, Department of Transportation, Washington, DC (1976).
5. System Design Analysis, Federal Aviation Administration Draft Advisory Circular, AFS-130, issued by Department of Transportation, Washington, DC (1977).
6. L. G. Stucki and G. L. Foshee, "New Assertion Concepts for Self-Metric Software Validation," Proceedings 1975 International Conference on Reliable Software, IEEE Cat. No. 75CH0940-7CSR, New York, NY (1975), pp. 59-71.

7. H. Hecht, Reliability Testing of the Fault Tolerant Spacecraft Computer, SAMSO-TR-74-259. The Aerospace Corporation, El Segundo, CA (1974).

APPENDIX A

FLOW OF PASCAL SCHEDULER



NOTE: The timer also generates a hardware interrupt when decremented to zero.

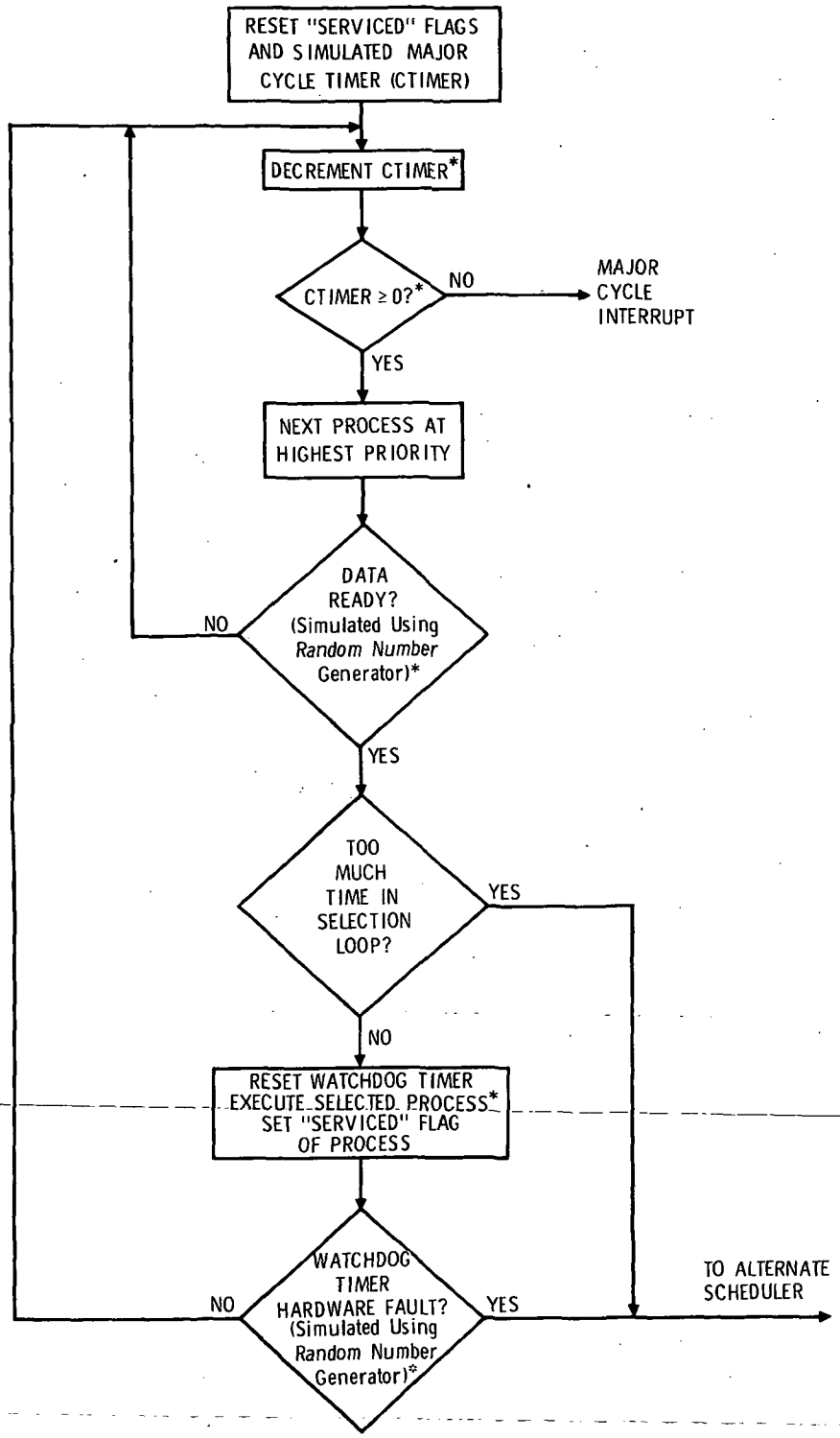
APPENDIX B

SCHEDULER SIMULATION

The simulation of the Fault-Tolerant Scheduler (FTS) was carried out on a CDC 7600 computer.

The system of programs consists of the Scheduler, Timerinterrupt, Alternatescheduler, Execute, and the test programs, Driver, Simerrors, and Set-readyflags. A top-level flow chart of the system is shown in Figure B-1. Scheduler contains both operational and test code. Timerinterrupt contains only operational code. Alternatescheduler has not been implemented. Alternatescheduler is to be called by Scheduler and Timerinterrupt to resume computation whenever either detects a catastrophic error. Execute is a program in name only; it has not been defined. It is included because it is called by Scheduler. The test program, Simerrors, will be used in its place.

Driver is used to initiate and control the duration of the test run. Driver starts the run by calling Scheduler. Upon being called, Scheduler attempts to select a task to be executed. If a task is not selected within a given time period, Scheduler calls Alternatescheduler. Scheduler will be called again or the run will be terminated by Driver according to the value of run control parameters. If successful, Scheduler will call Execute which will in turn call Simerrors.



*CODE INSERTED FOR SIMULATION PURPOSES

FIG. B-1 SCHEDULER SIMULATION LOGIC

Based on run parameters, Simerrors will simulate a normal or abnormal execution of the task; the latter to be detected either by Scheduler or Timerinterrupt. If the execution was normal, Scheduler attempts to select another task and the above process is repeated. If the execution was abnormal, an error message is output and the Alternatescheduler is called.* Scheduler will be recalled or the run terminated based on run-control parameters.

Setreadyflags sets flags to identify which tasks are to be executed. This function is normally done by the sensor when there is data to be processed. The task will be selected based on the state of the ready flag and the priority of the task.

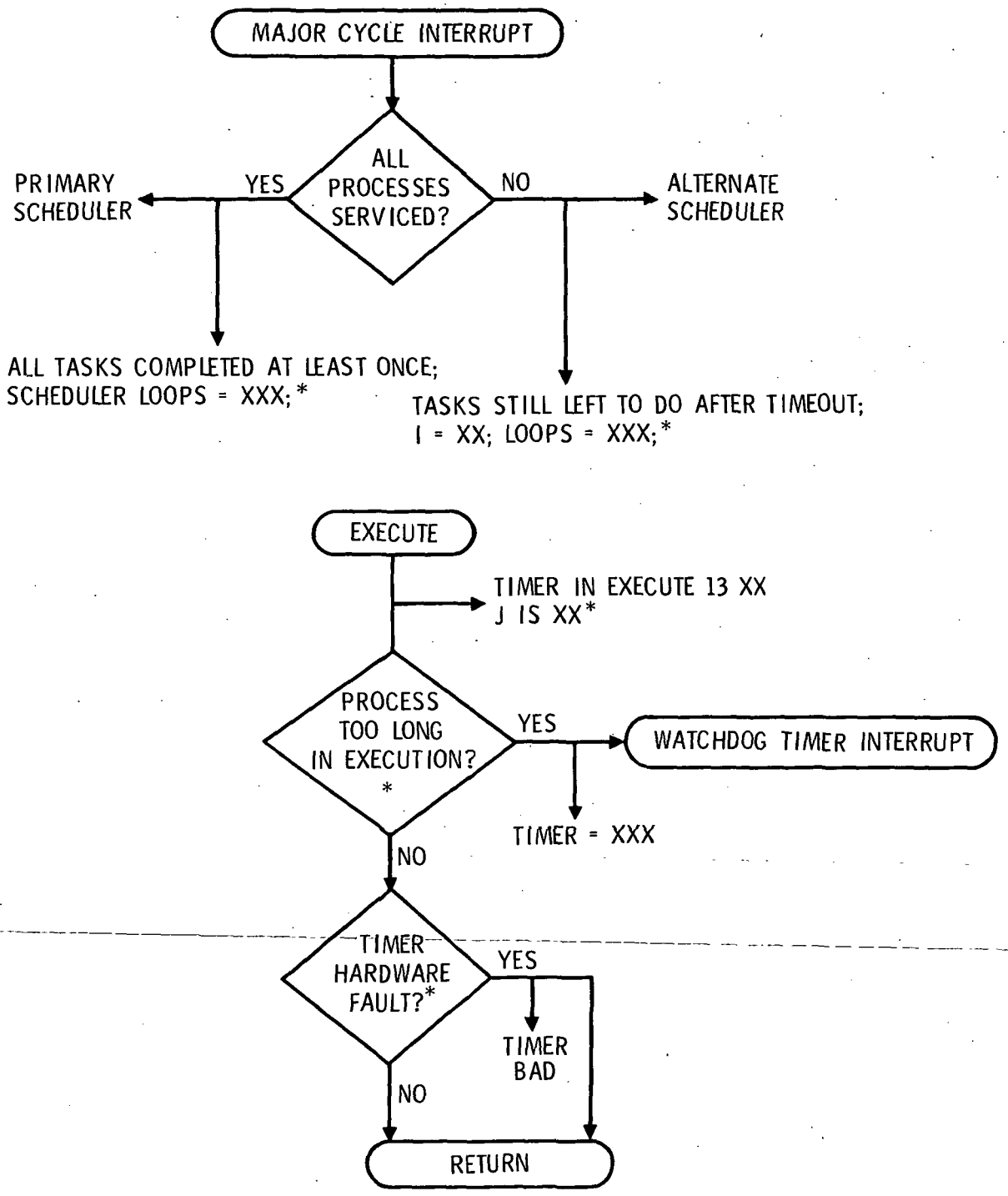
A random number generator is used to determine when an error is to occur and which task should be executed.

Figure B-2 contains the execution trace printed during a simulation run of the FTS. The simulation demonstrated that the scheduler and timer interrupt code executes, and that the software acceptance tests successfully detect timer failures. The simulation also provided insight into operation of the major cycle timer and the selection loop acceptance test, allowing us to vary the frequency which the alternate scheduler is called from these two points. Figure B-3 indicates those points in the simulation from which trace messages are written.

*Note: In full implementation, alternate processess would be invoked at this point instead of the Alternate-scheduler. Since no processes exist in this simulation, this call is substituted.

```
ALL  TASKS  COMPLETED  AT  LEAST  ONCE;  SCHEDULER  LOOPS  =          1
TIMER  IN  EXECUTE  IS  20  J  IS  9
TIMER  IN  EXECUTE  IS  20  J  IS  22
TIMER  IN  EXECUTE  IS  20  J  IS  3
TIMER  IN  EXECUTE  IS  20  J  IS  9
TIMER  IN  EXECUTE  IS  20  J  IS  9
TIMER  IN  EXECUTE  IS  20  J  IS  20
TIMER  IN  EXECUTE  IS  20  J  IS  8
TIMER  IN  EXECUTE  IS  20  J  IS  13
TIMER  IN  EXECUTE  IS  20  J  IS  10
TIMER  IN  EXECUTE  IS  20  J  IS  22
TIMER  IN  EXECUTE  IS  20  J  IS  5
TIMER  IN  EXECUTE  IS  20  J  IS  8
TIMER  IN  EXECUTE  IS  20  J  IS  13
TIMER  IN  EXECUTE  IS  20  J  IS  22
TIMER  IN  EXECUTE  IS  20  J  IS  18
TIMER  IN  EXECUTE  IS  20  J  IS  20
TIMER  IN  EXECUTE  IS  20  J  IS  5
TIMER  IN  EXECUTE  IS  20  J  IS  23
TIMER  IN  EXECUTE  IS  20  J  IS  9
TIMER  IN  EXECUTE  IS  20  J  IS  21
TIMER  IN  EXECUTE  IS  20  J  IS  19
TIMER  IN  EXECUTE  IS  20  J  IS  2
RNG  =  9.9965317416198F-001
TASKS  STILL  LEFT  TO  DO  AFTER  TIMEOUT  ;I  =  1  LOOPS  =          2
TIMER  FROM  ALTSCH  IS  0
```

FIG. B-2 SIMULATION TRACE OF FTS



*SIMULATED USING RANDOM NUMBER GENERATOR

FIG. B-3 PRINCIPAL SOURCES OF MESSAGES IN THE SIMULATION TRACE

GLOSSARY OF SOFTWARE HIERARCHY TERMS

The following explains terms that are used with more specialized meanings than in the general literature.

Application Program (or sometimes just Application) is a software segment that completely services an aircraft control function, e.g., attitude control program.

Process is a subdivision of an application program that yields a defined result (and is in the structure used here expected to execute without interruption), e.g., filtered attitude rate computation.

Recovery block is the basic structure for providing fault tolerance in a computer program. A recovery block consists of a primary routine and one or more alternate (or backup routines) and an acceptance test. The operation of a recovery block is explained in Section 2.

Degraded recovery block is one in which the alternate routine consists of a void return to the scheduler. (If the primary routine does not yield a result that passes the acceptance test, the function supported by the recovery block is not serviced.)

Routine is the application code (excluding acceptance test) for one alternate within a recovery block. A routine yields a result that can be subjected to an acceptance test. A routine consists of one or more processes.